1 HyperSpy User Guide

1.1 Introduction ............................................................. 1
1.2 What’s new ............................................................. 2
1.3 Changelog ............................................................... 3
1.4 Installing HyperSpy .................................................... 22
1.5 Getting started ......................................................... 25
1.6 Tools: the Signal class .................................................. 35
1.7 Signal1D Tools .......................................................... 66
1.8 Signal2D Tools .......................................................... 71
1.9 Data visualization ....................................................... 71
1.10 Machine learning ...................................................... 102
1.11 Model fitting ........................................................... 109
1.12 Electron Energy Loss Spectroscopy ................................ 129
1.13 Energy-Dispersive X-ray Spectrometry (EDS) ................... 132
1.14 Dielectric function tools .............................................. 147
1.15 Electron Holography .................................................. 148
1.16 Loading and saving data .............................................. 151
1.17 Events ................................................................. 165
1.18 Working with big data ................................................ 167
1.19 Metadata structure .................................................. 171
1.20 Bibliography .......................................................... 178

2 HyperSpy Developer Guide .............................................. 181

2.1 Introduction ........................................................... 181
2.2 Running and writing tests ........................................... 182
2.3 Writing documentation ............................................. 184
2.4 Coding style ......................................................... 185
2.5 Tips for writing methods that work on lazy signals ............ 185
2.6 Speeding up code .................................................... 186
2.7 Writing packages that extend HyperSpy ........................ 187

3 hyperspy ................................................................. 193

3.1 hyperspy package ................................................... 193

4 Credits ................................................................. 541

5 Citing HyperSpy ....................................................... 543

6 Indices and tables ..................................................... 545

Bibliography .............................................................. 547
<table>
<thead>
<tr>
<th>Python Module Index</th>
<th>549</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>553</td>
</tr>
</tbody>
</table>
1.1 Introduction

1.1.1 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. External packages can extend HyperSpy to e.g. implement features to analyse a particular sort of data. For details on how to register extensions see Writing packages that extend HyperSpy. For a list of packages that extend HyperSpy follow this link.

Note: From version 2.0, HyperSpy will be split into a core package (HyperSpy) that will provide the common infrastructure and a number of HyperSpy extensions specialized in the analysis of different types of data.

1.1.2 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 HyperSpy Developer Guide 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.
1.2 What’s new

1.2.1 Current Version

v1.5

NEW

• New method `hyperspy.component.Component.print_current_values()`. See the User Guide for details.

• New `hyperspy._components.skew_normal.SkewNormal` component.


• Estimation of number of significant components by the elbow method. See Scree plots.

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.drawing.utils.plot_images()` or `hyperspy.drawing.utils.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`
  - `hyperspy._components.volume_plasmon_drude.VolumePlasmonDrude`
  - `hyperspy._components.eels_double_power_law.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 the `hyperspy._components.polynomial_deprecated.Polynomial` component constructor.
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.

1.3 Changelog

1.3.1 Previous Versions

We only cover here the main highlights, for a detailed list of all the changes see the commits in the GITHUB milestones.

v1.4.2

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.

v1.4.1

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.

v1.4

This is a minor release. Follow the following links for details on all the bugs fixed, enhancements and new features.

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 SPX format.
  - Reading and writing the mrcz open format. See MRCZ.
- New 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 FFT and iFFT.
- New statistics() method to compute useful hologram parameters. See Getting hologram statistics.
- 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.
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 interatively 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 FFT and iFFT.

- The `align2D()` and `estimate_shift2D()` can operate with sub-pixel accuracy using skimage’s upsampled matrix-multiplication DFT. See Two dimensional signal registration (alignment).

v1.3.2

This is a maintenance release. Follow the following links for details on all the bugs fixed and enhancements.

v1.3.1

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

v1.3

This is a minor release. Follow the following links for details on all the bugs fixed, feature and documentation enhancements, and new features.

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.
Enhancements

- Creating many markers is now much faster.
- New “Stage” metadata node. See Metadata structure for details.
- The Brucker file reader now supports the new version of the format. See Bruker composite file.
- 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 make 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

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
  - show(). Use gui() instead.
  - All notebook_interaction() method. Use the equivalent gui() method instead.
  - integrate_in_range(). Use integrate1D() instead.
- The following items have been removed from preferences:
  - General.default_export_format
  - General.lazy
  - Model.default_fitter
For developers

- In addition to adding ipywidgets GUI elements, the traitsui GUI elements have been split into a separate package. See the new hyperspy_gui_traitsui package.
- The new `ui_registry` enables easy connection of external GUI elements to HyperSpy. This is the mechanism used to split the traitsui and ipywidgets GUI elements.

**v1.2**

This is a minor release. Follow the following links for details on all the bugs fixed, enhancements and new features.

**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 functionality). See *Iterating external functions with the map method*.
- `Electron Holography` reconstruction.
- Support for reading *EDAX TEAM SPD and SPC* files.
- New signal methods `indexmin()` and `valuemin()`.

**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*.
HyperSpy Documentation, Release 1.5.1.dev

- `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 *load-multiple-label*.
- 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 *hdf5-format*. This will be the default extension in HyperSpy 1.3.

**For developers**

- Most of HyperSpy plotting features are now covered by unitests. See *Plot testing*.
- unitests migrated from nose to pytest. See tests-label.

**v1.1.2**

This is a maintenance release. Follow the following links for details on all the *bugs fixed* and *enhancements*.

**v1.1.1**

This is a maintenance release. Follow the following link for details on all the *bugs fixed*.

**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.

**v1.1**

This is a minor release. Follow the following links for details on all the *bugs fixed*.

**NEW**

- *Transposing (changing signal spaces)*.
- *Protochips log* reader.
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.

v1.0.1

This is a maintenance release. Follow the following links for details on all the bugs fixed.

v1.0

This is a major release. Here we only list the highlist. A detailed list of changes is available in github.

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 spectra* from The EELS Database.
- *Interactive operations*.
- *Events*.

Model

- *Smart Adaptive Multi-dimensional Fitting (SAMFire).*
- Store models in hdf5 files.
- Add *fancy indexing* to *Model*.
- Two-dimensional model fitting.

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.

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.

Documentation

- The following sections of the User Guide were revised and largely overwritten:
  - *Installing HyperSpy*.
  - *Machine learning*.
  - *Energy-Dispersive X-ray Spectrometry (EDS)*.
- New *HyperSpy Developer Guide*.

API changes

- Split `components` into `components1D` and `components2D`.
- Remove `record_by` from metadata.
- Remove simulation classes.
- The `Signal1D`, `Signal2D` and `BaseSignal` classes deprecated the old `Spectrum Image` and `Signal` classes.

v0.8.5

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.

New feature

- Widgets to interact with the model components in the Jupyter Notebook. See *here* and #1007.
API changes

The new `BaseSignal`, `Signal1D` and `Signal2D` deprecate `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.

v0.8.4

This release adds support for Python 3 and drops support for Python 2. In all other respects it is identical to v0.8.3.

v0.8.3

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

v0.8.2

This is a maintenance release that fixes an issue with the Python installers. Those who have successfully installed v0.8.1 do not need to upgrade.

v0.8.1

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.

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.
- ` unfolded()` context manager.
- ` derivative()` method.
- ` syntax to access the components in the model` that includes pretty printing of the components.
API changes

- **hspy** is now deprecated in favour of the new **api**. The new API renames and/or move several modules as follows:
  - **hspy.components** -> **api.model.components**
  - **hspy.utils** -> **api**
  - **hspy.utils.markers api.plot.markers**
  - **hspy.utils.example_signals** -> **api.datasets.example_signals**

In HyperSpy 0.8.1 the full content of **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 **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 **Signal** class is now deprecated. We encourage all users to use **isig** and **inav** instead for indexing.
- **create_model()** is now deprecated in favour of the new equivalent **create_model() Signal** method.
- **unfold_if_multidim()** is deprecated.

v0.8

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 (see *hdf5-format* )

Plotting

- New class, **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 **plot()** method to customize the image. See *Customising image plot*.

EDS

- New method for quantifying EDS TEM spectra using Cliff-Lorimer method, **quantification()**. See *EDS Quantification*.

1.3. Changelog
• New method to estimate for background subtraction, `estimate_background_windows()`. See *Background subtraction*.

• New method to estimate the windows of integration, `estimate_integration_windows()`.

• New specific `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.
  
  - `load_1D_EDS_SEM_spectrum()`
  
  - `load_1D_EDS_TEM_spectrum()`

• New method to mask the vacuum, `vacuum_mask()` and a specific `decomposition()` method that incorporate the vacuum mask

**API changes**

• `Component` and `Parameter` now inherit `traits.api.HasTraits` that enable `traitsui` to modify these objects.

• `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`

**v0.7.3**

This is a maintenance release. A list of fixed issues is available in the 0.7.3 milestone in the github repository.

**v0.7.2**

This is a maintenance release. A list of fixed issues is available in the 0.7.2 milestone in the github repository.

**v0.7.1**

This is a maintenance release. A list of fixed issues is available in the 0.7.1 milestone in the github repository.

**New features**

• Add suspend/resume model plot updating. See *Visualizing the model*. 

New features

Core

- New syntax to index the `AxesManager`.
- New Signal methods to transform between Signal subclasses. More information [here](#).
  - `set_signal_type()`
  - `set_signal_origin()`
  - `as_signal2D()`
  - `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 GTK and TK toolkits, although with no GUI features.
- It is now possible to run HyperSpy in a headless system.
- Added a CLI to `remove_background()`.
- New `estimate_peak_width()` method to estimate peak width.
- New methods to integrate over one axis: `integrate1D()` and `integrate_in_range()`.
- New metadata attribute, `Signal.binned`. Several methods behave differently on binned and unbinned signals. See *Binned and unbinned signals*.
- New `map()` method to easily transform the data using a function that operates on individual signals. See *Iterating over the navigation axes*.
- New `get_histogram()` and `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 `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 plot() method to customize the navigator. See Data visualization.
- Add compatibility with IPython’s matplotlib inline plotting. See 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, 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 Signal instances. See Obtaining the results as BaseSignal instances.
- New methods to plot the decomposition and blind source separation results that support n-dimensional loadings. See Visualizing results.

Dielectric function

- New Signal subclass, DielectricFunction.

EELS

- New method, 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, align_zero_loss_peak().

EDS

- New signal, EDSSpectrum especialized in EDS data analysis, with subsignal for EDS with SEM and with TEM: EDSSSEMSpectrum 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, `add_elements()` and `add_lines()`. See *Describing the sample*

• New method to get the intensity of specific X-ray lines: `get_lines_intensity()`. See `eds_plot-label`

### 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:
  - `swap_axes()`
  - `rebin()`

### v0.6

#### 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: `estimate_zero_loss_peak_centre()`, `estimate_elastic_scattering_intensity()` and `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 `get_fine_structure_as_spectrum()` EELSCLEdge method.

• New Arctan model component.

• New `enable_adjust_position()` and `disable_adjust_position()` to easily change the position of components using the mouse on the plot.

• New Model methods `set_parameters_value()`, `set_parameters_free()` and `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: `integrate_simpson()`, `max()`, `min()`, `var()`, and `std()`.
• New sliders window to easily navigate signals with navigation_dimension > 2.
• The Ripple (rpl) reader can now read rpl files produced by INCA.

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)})

v0.5.1

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`. 
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.

API changes

- `EELSSPectrum.find_low_loss_centre` was renamed to `estimate_zero_loss_peak_centre`.
- `EELSSPectrum.calculate_FWHM` was renamed to `estimate_FWHM`.

v0.5

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 `sigmal3` 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.
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 constrast of the image was not automatically updated.

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.

v0.4.1

New features

- Added TIFF 16, 32 and 64 bits support by using (and distributing) Christoph Gohlke’s tifffile 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.

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 unmantained and broken 2D peak finding and analysis features.

Syntax changes

• In EELS automatic background feature creates a PowerLaw component, adds it to the model an 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.

v0.4

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 automically updated on chaning the free attribute.

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 calibration UIs were freezing when the plots were 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.

Syntax changes

• in decomposition V rename to explained_variance.
• In FixedPattern, default interpolation changed to linear.
• Line and parabola 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 reflect the last changes.
• The microscopes.csv file was updated so it no longer contains the Orsay VG parameters.

1.4 Installing HyperSpy

The easiest way to install HyperSpy in Microsoft Windows is installing the HyperSpy Bundle.

For quick instructions on how to install HyperSpy in Linux, MacOs or Windows using the Anaconda Python distribution see Quick instructions to install HyperSpy using Anaconda (Linux, MacOs, Windows).

Those experienced with Python may like to Install using Python installers or Install from source.

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.

1.4.1 HyperSpy Bundle for Microsoft Windows

The easiest way to install HyperSpy in Windows is installing the HyperSpy Bundle. This is a customised WinPython distribution that includes HyperSpy, all its dependencies and many other scientific Python packages.

For details and download links go to https://github.com/hyperspy/hyperspy-bundle

1.4.2 Quick instructions to install HyperSpy using Anaconda (Linux, MacOs, Windows)

Anaconda is recommended for the best performance (it is compiled using Intel MKL libraries) and the easiest installation. The academic license is free.

1. Download and install Anaconda. If you are not familiar with Anaconda please refer to their User Guide for details.

2. Then install HyperSpy executing the following conda commands in the Anaconda Prompt, Linux/Mac Terminal or Microsoft Windows Command Prompt. (This depends on your OS and how you have installed Anaconda, see the Anaconda User Guide) for details.

   $ conda install hyperspy -c conda-forge

3. (optional) Since HyperSpy v1.3 the traitsui GUI elements are not installed automatically (but the Jupyter GUI elements are). To install them:

   $ conda install hyperspy-gui-traitsui -c conda-forge
Note: Since version 0.8.4 HyperSpy only supports Python 3. If you need to install HyperSpy in Python 2.7 install version 0.8.3:

$ conda install traitsui
$ pip install --upgrade hyperspy==0.8.3-1

To enable context-menu (right-click) startup in a chosen folder, install start_jupyter_cm. (Currently only available for Gnome and Windows, not MacOS.)

For more options and details read the rest of the documentation.

### 1.4.3 Install using Python installers

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.

Install using pip:

$ pip install hyperspy

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

$ pip install --upgrade hyperspy==0.8.3-1

pip installs automatically the strictly required libraries. However, for full functionality you may need to install some other dependencies. To install with full functionality:

$ pip install hyperspy[all]

Alternatively you can select the extra functionalities required:

- learning to install required libraries for some machine learning features.
- gui-jupyter to install required libraries to use the Jupyter widgets GUI elements.
- gui-traitsui to install required libraries to use the GUI elements based on traitsui
- test to install required libraries to run HyperSpy’s unit tests.
- mrcz to install the mrcz plugin.
- build-doc to install required libraries to build HyperSpy’s documentation.
- speed install optional libraries that speed up some functionalities.

For example:

$ pip install hyperspy[learning, gui-jupyter]

See also *Installing the required libraries*.

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. If the above does not work for you remember that the easiest way to install HyperSpy is *using Anaconda*. 

### 1.4. Installing HyperSpy
1.4.4 Install from a binary

We provide binary distributions for Windows (see the Downloads section of the website). To install easily in other platforms see Install using Python installers

1.4.5 Install from source

Released version

To install from source grab a tar.gz release and in Linux/Mac (requires to Installing the required libraries manually):

```bash
$ tar -xzf hyperspy.tar.gz
$ cd hyperspy
$ python setup.py install
```

You can also use a Python installer, e.g.

```bash
$ pip install hyperspy.tar.gz
```

Development version

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

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

To install HyperSpy you could proceed like in Released version. However, if you are installing from the development version most likely you will prefer to install HyperSpy using pip development mode:

```bash
$ cd hyperspy
$ pip install -e ./
```

All required dependencies are automatically installed by pip. However, for extra functionality you may need to install some extra dependencies, see Installing the required libraries. If using your system python distribution, it is recommended to install using the --user flag:

```bash
$ cd hyperspy
$ pip install -e --user ./
```

Creating Debian/Ubuntu binaries

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

```bash
$ ./release_debian
```

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

In addition to the libraries that are automatically installed when installing HyperSpy using *pip* (see *Install using Python installers*).

In case some of the required libraries are not automatically installed when installing from source in a conda environment, these can be obtained beforehand by installing and removing hyperspy from that environment;

1.4.7 Known issues

Windows

- If HyperSpy fails to start in Windows try installing the Microsoft Visual before reporting a bug.
- If HyperSpy raises a MemoryError exception:
  - Install the 64bit version if you’re using the 32bit one and you are running HyperSpy in a 64bit system.
  - Increase the available RAM by closing other applications or physically adding more RAM to your computer.

1.5 Getting started

1.5.1 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.

1.5.2 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:

```bash
$ 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.

1.5.3 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.):
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:

```python
>>> 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.

**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 is 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.

By default, HyperSpy warns the user if one of the GUI packages is not installed. These warnings can be turned off using the preferences GUI or programmatically as follows:

```python
>>> import hyperspy.api as hs
>>> hs.preferences.GUIs.warn_if_guis_are_missing = False
>>> hs.preferences.save()
```

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.

**Warning:** When using the qt4 backend in Python 2 the matplotlib magic must be executed after importing HyperSpy and qt must be the default HyperSpy backend.

**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:

```python
>>> import matplotlib
>>> matplotlib.rcParams["backend"] = "Agg"
>>> import hyperspy.api as hs
```
1.5.4 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.:

```python
>>> hs?
>>> hs.load?
>>> hs.signals?
```

This syntax is a shortcut to the standard way 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.

Please note that the documentation of the code is a work in progress, so not all the objects are documented yet. Up-to-date documentation is always available in the HyperSpy website.

1.5.5 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 documentation (specially their Getting started section) for many more useful features that will boost your efficiency when working with HyperSpy/Python interactively.

1.5.6 Loading data

Once HyperSpy is running, to load from a supported file format (see Supported formats) simply type:

```python
>>> s = hs.load("filename")
```

**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.:

```python
>>> # 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 files: the load function

1.5.7 “Loading” data 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.
The numpy array is stored in the data attribute of the signal class.

## 1.5.8 Loading example data and data from online databases

HyperSpy is distributed with some example data that can be found in `hs.datasets.example_signals`. The following example plots one of the example signals:

```python
>>> hs.datasets.example_signals.EDS_TEM_Spectrum().plot()
```

New in version 1.4: *artificial_data*

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

```python
>>> s = hs.datasets.artificial_data.get_core_loss_eels_signal()
>>> s.plot()
```

The `eelsdb()` function in `hs.datasets` can directly load spectra from The EELS Database. For example, the following loads all the boron trioxide spectra currently available in the database:

```python
>>> hs.datasets.eelsdb(formula="B2O3")
[<EELSSpectrum, title: Boron oxide, dimensions: (520)>,
 <EELSSpectrum, title: Boron oxide, dimensions: (520)>]
```

## 1.5.9 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.

```python
>>> 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.

```python
>>> 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 \((X, Y)\) 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: \((E, D \mid X, Y)\).

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:

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.

### 1.5.10 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.

```python
>>> s = hs.signals.Signal1D(np.random.random((10, 20 , 100)))

>>> s
<Signal1D, title: , dimensions: (20, 10|100)>

>>> s.axes_manager
<Axes manager, axes: (<Unnamed 0th axis, size: 20, index: 0>, <Unnamed 1st axis, size: 10, index: 0>|<Unnamed 2nd axis, size: 100>)>

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

The axis properties can be set by setting the `DataAxis` attributes e.g.

```python
>>> 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.:

```python
>>> 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`
Fig. 1: AxesManager ipywidgets GUI.

or the `DataAxis`, e.g:

Fig. 2: DataAxis ipywidgets GUI.

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

Alternatively, the “current position” can be changed programmatically by directly accessing the `indices` attribute of a Signal’s `AxesManager`. This is particularly useful if trying to set a specific location with which to initialize a model’s parameters to sensible values before preforming 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:
1.5.11 Using quantity and converting units

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

```
>>> s = hs.signals.Signal1D(np.arange(10))
>>> s.axes_manager[0].scale_as_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 | 2.5 | µm
>>> s.axes_manager[0].offset_as_quantity = '2.5 nm'
>>> s.axes_manager
<Axes manager, axes: (|10)>
Name | size | index | offset | scale | units
----- | ----- | ------ | ------ | ------ | -------
<undefined> | 10 | | 2.5 | 2.5e+03 | 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
pint.quantity.build_quantity_class.<locals>.Quantity
>>> q
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 units to avoid using too large or small number.

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

```
>>> axis = hs.hyperspy.axes.DataAxis(size=10, scale=0.1, offset=10, units='mm')
>>> axis.scale_as_quantity
0.1 millimeter
>>> axis.convert_to_units('µm')
>>> axis.scale_as_quantity
100.0 micrometer
```
1.5.12 Saving Files

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

```python
>>> # 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 format in HyperSpy is hspy, which is based on the HDF5 format. This format keeps the most information possible.

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

1.5.13 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.

```python
>>> s = hs.load("NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217.msa")
>>> s.metadata
original_filename = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217.msa
record_by = spectrum
signal_type = EELS
title = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217

>>> s.original_metadata
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 =

>>> s.set_microscope_parameters(100, 10, 20)
>>> s.metadata
TEM
EELS
collection_angle = 20
beam_energy = 100
convergence_angle = 10
original_filename = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217.msa
record_by = spectrum
signal_type = EELS
title = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217
```

(continues on next page)
1.5.14 Configuring HyperSpy

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

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

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

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:

```python
>>> 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.

1.5.15 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.:

```python
>>> import hyperspy.api as hs
>>> hs.set_log_level('INFO')
>>> hs.load(r'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])
```
Fig. 4: Preferences user interface.
1.6 Tools: the Signal class

1.6.1 The Signal class and its subclasses

**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 `EELSSpectrum` (which is a subclass of `Signal1D`) adds extra functionality to the `Signal1D` class for electron energy-loss spectroscopy data analysis.

The table below summarises all the specialised `BaseSignal` subclasses currently distributed with HyperSpy.

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:

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

The different signals 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.

1.6.2 Signal initialization

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>size</th>
<th>index</th>
<th>offset</th>
<th>scale</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis1</td>
<td>20</td>
<td></td>
<td>2</td>
<td>0.1</td>
<td>B</td>
</tr>
<tr>
<td>Axis0</td>
<td>10</td>
<td></td>
<td>1</td>
<td>0.2</td>
<td>A</td>
</tr>
</tbody>
</table>
```

This also applies to the metadata.

```python
>>> 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
  name = A BaseSignal
  title = A BaseSignal title
Signal
  binned = False
  signal_type =
```

1.6. Tools: the Signal class

35
1.6.3 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.:

```python
>>> 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.:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.

### Transforming between signal subclasses

The different subclasses are characterized by the `signal_type` metadata attribute, the data `dtype` and the signal dimension. See the table and diagram below. `signal_type` describes the nature of the signal. It can be any string, normally the acronym associated with a particular signal. In certain cases HyperSpy provides features that are only available
for a particular signal type through `BaseSignal` subclasses. The `BaseSignal` method `set_signal_type()` changes the signal_type in place, which may result in a `BaseSignal` subclass transformation.

Furthermore, the `dtype` of the signal data also affects the subclass assignment. There are e.g. specialised signal subclasses to handle complex data (see the following diagram).

![Diagram showing the inheritance structure of the different subclasses](image)

---

**Table 1:** `BaseSignal` subclass metadata attributes.

<table>
<thead>
<tr>
<th>BaseSignal subclass</th>
<th>signal_dimension</th>
<th>signal_type</th>
<th>dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BaseSignal</code></td>
<td>•</td>
<td>•</td>
<td>real</td>
</tr>
<tr>
<td><code>Signal1D</code></td>
<td>1</td>
<td>•</td>
<td>real</td>
</tr>
<tr>
<td><code>EELSSpectrum</code></td>
<td>1</td>
<td>EELS</td>
<td>real</td>
</tr>
<tr>
<td><code>EDSSEMSpectrum</code></td>
<td>1</td>
<td>EDS_SEM</td>
<td>real</td>
</tr>
<tr>
<td><code>EDSTEM</code></td>
<td>1</td>
<td>EDS_TEM</td>
<td>real</td>
</tr>
<tr>
<td><code>Signal2D</code></td>
<td>2</td>
<td>•</td>
<td>real</td>
</tr>
<tr>
<td><code>HologramImage</code></td>
<td>2</td>
<td>hologram</td>
<td>real</td>
</tr>
<tr>
<td><code>DielectricFunction</code></td>
<td>1</td>
<td>DielectricFunction</td>
<td>complex</td>
</tr>
<tr>
<td><code>ComplexSignal</code></td>
<td>•</td>
<td>•</td>
<td>complex</td>
</tr>
<tr>
<td><code>ComplexSignal1D</code></td>
<td>1</td>
<td>•</td>
<td>complex</td>
</tr>
<tr>
<td><code>Complex2D</code></td>
<td>2</td>
<td>•</td>
<td>complex</td>
</tr>
</tbody>
</table>
New in version 1.5: External packages can register extra `BaseSignal` subclasses.

Note that, if you have packages that extend HyperSpy installed in your system, there may be more specialised signals available to you. To print all available specialised `BaseSignal` subclasses installed in your system call the `hyperspy.utils.print_known_signal_types()` function as in the following example:

```python
>>> hs.print_known_signal_types()
+--------------------+---------------------+--------------------+----------+
| signal_type | aliases | class name | package |
+--------------------+---------------------+--------------------+----------+
| DielectricFunction | dielectric function | DielectricFunction | hyperspy |
| EDS_SEM | | EDSSEMSpectrum | hyperspy |
| EDS_TEM | | EDSTEMSpectrum | hyperspy |
| EELS | TEM EELS | EELSSpectrum | hyperspy |
| hologram | | HologramImage | hyperspy |
| MySignal | | MySignal | hspy_ext |
+--------------------+---------------------+--------------------+----------+
```

**Warning:** From version 2.0 HyperSpy will no longer ship `BaseSignal` subclasses that are specific to a particular type of data (i.e. with non-empty `signal_type`). All those signals currently distributed with HyperSpy will be moved to new packages.

The following example shows how to transform between different subclasses.

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

### 1.6.4 Binned and unbinned signals

Signals that are a histogram of a probability density function (pdf) should have the `signal.metadata.Signal.binned` attribute set to True. This is because some methods operate differently in signals that are *binned*.

Changed in version 1.0: `Simulation`, `SpectrumSimulation` and `ImageSimulation` classes removed.

The default value of the `binned` attribute is shown in the following table:
Table 2: Binned default values for the different subclasses.

<table>
<thead>
<tr>
<th>BaseSignal subclass</th>
<th>binned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseSignal</td>
<td>False</td>
</tr>
<tr>
<td>Signal1D</td>
<td>False</td>
</tr>
<tr>
<td>EELSSpectrum</td>
<td>True</td>
</tr>
<tr>
<td>EDSSEMSpectrum</td>
<td>True</td>
</tr>
<tr>
<td>EDSTEM</td>
<td>True</td>
</tr>
<tr>
<td>Signal2D</td>
<td>False</td>
</tr>
<tr>
<td>ComplexSignal</td>
<td>False</td>
</tr>
<tr>
<td>ComplexSignal1D</td>
<td>False</td>
</tr>
<tr>
<td>Complex2Dmixin</td>
<td>False</td>
</tr>
</tbody>
</table>

To change the default value:

```python
>>> s.metadata.Signal.binned = True
```

1.6.5 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.

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:
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:

```python
>>> 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:

```python
>>> 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 ufuncs) that operate on numpy arrays can also operate on `BaseSignal` instances, however they return a numpy array instead of a `BaseSignal` instance e.g.
HyperSpy Documentation, Release 1.5.1.dev

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.
  - 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:

```python
>>> 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)>
>>> 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]
```

(continues on next page)
Unlike numpy, HyperSpy supports indexing using decimal numbers or string (containing a decimal number and an units), in which case HyperSpy indexes using the axis scales instead of the indices.

```python
>>> 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])
```
HyperSpy Documentation, Release 1.5.1.dev

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

```python
>>> 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 | 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]])
1.6. Tools: the Signal class
```
The same syntax can be used to set the data values in signal and navigation dimensions respectively:

```python
>>> 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])
>>> 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])
```

### Signal operations

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

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.
```python
>>> 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.

```python
>>> 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:

```python
>>> 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)>
>>> 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:

```python
>>> 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)
```

1.6. Tools: the Signal class
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:

```python
>>> image_stack = hs.signals.Signal2D(np.random.random((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:

```python
>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.misc.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)
>>> # 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. 6: Rotation of images by iteration.](image)

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.misc.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. 7: 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.misc.ascent()]*4))
>>> image_stack.axes_manager[1].name = "x"
>>> 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. 8: 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). As such, most HyperSpy functions cannot operate on such Signal, and the data should be accessed directly.

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).

```python
>>> import scipy.ndimage
>>> image_stack = hs.signals.Signal2D(np.array([scipy.misc.ascent()]*4))
>>> angles = hs.signals.BaseSignal(np.array([0, 45, 90, 135]))
>>> result = image_stack.map(scipy.ndimage.rotate,
...    angle=angles.T,
...    inplace=False,
...    reshape=True)
100%|| 4/4 [00:00<00:00, 18.42it/s]
```

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

New in version 1.2.0: `parallel` keyword. The execution can be sped up by passing `parallel` keyword to the `map()` method:

```python
>>> import time
>>> def slow_func(data):
...    time.sleep(1.)
...    return data + 1
>>> s = hs.signals.Signal1D(np.arange(20).reshape((20,1)))
>>> s
<Signal1D, title: , dimensions: (20|1)>
>>> s.map(slow_func, parallel=False)
100%|| 20/20 [00:20<00:00, 1.00s/it]
>>> # some operations will be done in parallel:
>>> s.map(slow_func, parallel=True)
100%|| 20/20 [00:02<00:00, 6.73it/s]
```

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`.

```python
>>> s = hs.signals.Signal1D(np.random.rand(10, 512))
>>> d = hs.signals.Signal1D(np.cos(np.linspace(0., 2*np.pi, 512)))
>>> s.map(lambda A, B: A * B, B=d)
100%|| 10/10 [00:00<00:00, 2573.19it/s]
```

### 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.
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()` method 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 and is generally slower if Numba is not installed.

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

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum().as_lazy()
>>> print(s)
<LazyEDSSEMSpectrum, title: EDS SEM Spectrum, dimensions: (|1024)>
>>> print(s.rebin(scale=[2]))
<LazyEDSSEMSpectrum, title: EDS SEM Spectrum, dimensions: (|512)>
```

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

```python
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDSTEMSpectrum, title: , dimensions: (4, 4|10)>
>>> print('Sum =', spectrum.data.sum())
Sum = 164.0
>>> scale = [0.5, 0.5, 5]
>>> test = spectrum.rebin(scale=scale)
>>> print(test)
<EDSTEMSpectrum, title: , dimensions: (8, 8|2)>
>>> print('Sum =', test.data.sum())
Sum = 164.0
>>> print('Sum =', test2.data.sum())
Sum = 164.0
>>> spectrum.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 original signal shape
```

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

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

**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.

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

![Stacking example](image)

Fig. 9: Stacking example.

An object can be split into several objects with the `split()` method. This function can be used to reverse the `stack()` function:
HyperSpy Documentation, Release 1.5.1.dev

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

>>> import numpy as np
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft_power = np.log(im.fft(shift=True).amplitude)
>>> fft_power_apodized = np.log(im.fft(shift=True, apodization=True).amplitude)
>>> hs.plot.plot_images([fft_power, fft_power_apodized], tight_layout=True)

apodization attribute can also take following values which correspond to types of apodization windows: hann (or apodization=True), hamming, tukey.

Note that for visual inspection of FFT it is common to plot logarithm of amplitude rather than FFT itself as it is done in the example above.

By default both methods calculate FFT and IFFT with origin at (0, 0) (not in the centre of FFT). Use shift=True option to calculate FFT and the inverse with origin shifted in the centre.

>>> im_ifft = im.fft(fft_shift=True).ifft(fft_shift=True)

FFT and iFFT

The Fast Fourier transform and its inverse can be applied on a signal with the \texttt{fft()} and the \texttt{ifft()} methods. In order to remove streaks in FFT (usually used only for presenting FFT patterns rather than for quantitative analyses) use \texttt{apodization} attribute as follows:

>>> import numpy as np
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft_power = np.log(im.fft(shift=True).amplitude)
>>> fft_power_apodized = np.log(im.fft(shift=True, apodization=True).amplitude)
>>> hs.plot.plot_images([fft_power, fft_power_apodized], tight_layout=True)

apodization attribute can also take following values which correspond to types of apodization windows: hann (or apodization=True), hamming, tukey.

Note that for visual inspection of FFT it is common to plot logarithm of amplitude rather than FFT itself as it is done in the example above.

By default both methods calculate FFT and IFFT with origin at (0, 0) (not in the centre of FFT). Use \texttt{shift=True} option to calculate FFT and the inverse with origin shifted in the centre.

>>> im_ifft = im.fft(fft_shift=True).ifft(fft_shift=True)
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.:

```python
>>> 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
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) 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.

```python
>>> rgb_test = np.zeros((1024, 1024, 3))
>>> ly, lx = rgb_test.shape[:2]
>>> offset_factor = 0.16
>>> size_factor = 3
(continues on next page)```
>>> 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()
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 is swaps the signal and navigation axes. For example:

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

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

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

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.

```python
>>> # 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:

```python
>>> s2 = hs.signals.BaseSignal(np.random.rand(2, 2)) # 2D signal
>>> s3 = hs.signals.BaseSignal(np.random.rand(3, 3)) # 3D signal
>>> s4 = hs.signals.BaseSignal(np.random.rand(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.:

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

```python
>>> 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'`.)

```python
>>> im = hs.datasets.example_signals.reference_hologram().isig[:200, :200]
>>> ima = im.apply_apodization(window='hann', hann_order=3)
>>> hs.plot.plot_images([im, ima], vmax=3000, tight_layout=True)
```

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.

### 1.6.6 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:

```python
>>> s = hs.signals.EELSSpectrum(np.random.normal(size=(10,100)))
>>> s.print_summary_statistics()
Summary statistics
------------------
mean:  0.021
std:   0.957
min:  -3.991
Q1:  -0.608
median:  0.013
Q3:  0.652
max:   2.751

>>> 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:

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

### 1.6.7 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:

```python
s.metadata.Signal.set_item("Noise_properties.variance", 10)
```
For heterocedastic noise the variance attribute must be a `BaseSignal`. Poissonian noise is a common case of heterocedastic noise where the variance is equal to the expected value. The `estimate_poissonian_noise_variance()` `BaseSignal` 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 then 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:

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

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:

```python
>>> s = hs.signals.Spectrum(np.ones(100))
>>> s.add_poissonian_noise()
>>> s.metadata
```

(continues on next page)
>>> 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 = <SpectrumSimulation, title: Variance of , dimensions: (|100)>
               binned = False
               signal_type =

1.6.8 Speeding up operations

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.

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

1.6.9 Interactive operations

The function interactive() ease the task of defining operations that are automatically updated when an event is triggered. By default it recomputes the operation when data or the axes of the original signal changes.

```python
>>> s = hs.signals.Signal1D(np.arange(10.))
>>> ssum = hs.interactive(s.sum, axis=0)
>>> ssum.data
array([45.0])
>>> s.data /= 10
>>> s.events.data_changed.trigger(s)
>>> ssum.data
array([ 4.5])
```
The interactive operations can be chained.

```python
>>> 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.]])
```

1.6.10 Region Of Interest (ROI)

A number of different ROIs are available:

- `Point1DROI`
- `Point2DROI`
- `SpanROI`
- `RectangularROI`
- `CircleROI`
- `Line2DROI`

Once created, a ROI can be used to return a part of any compatible signal:

```python
>>> 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 examples 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()`.

```python
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.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()
```

(continues on next page)
Notably, since ROIs are independent from the signals they sub-select, the widget can be plotted on a different signal altogether.

```python
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> s = hs.signals.Signal1D(np.random.rand(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.
1.6. Tools: the Signal class
(0, 0)
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.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,
... any_axis_changed,
... 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.

ROIs can be used in place of slices when indexing and to define a signal range in functions taken a `signal_range` argument. For example:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> roi = hs.roi.SpanROI(left=5, right=15)
>>> sc = s.isig[roi]
>>> s.remove_background(signal_range=roi, background_type="Polynomial")
>>> im = hs.datasets.example_signals.object_hologram()
>>> roi = hs.roi.RectangularROI(left=120, right=460., top=300, bottom=560)
>>> imc = im.isig[roi]
```

New in version 1.3: `gui()` method.

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:

```python
>>> # 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`:

```python
>>> import scipy
>>> holo = hs.datasets.example_signals.object_hologram()
```

(continues on next page)
>>> roi = hs.roi.Line2DROI(x1=465.577, y1=445.15, x2=169.4, y2=387.731, linewidth=0)
>>> holo.plot()
>>> ss = roi.interactive(holo)

By default output of the method is in degrees, though radians can be selected as follows:

>>> roi.angle(axis='y', units='radians')
-1.7622880506791903

Conveniently, `angle()` can be used to rotate image to align selected features with respect to vertical or horizontal axis:

>>> holo.map(scipy.ndimage.rotate, angle=roi.angle(axis='horizontal'), inplace=False).
--plot()
1.6.11 Handling complex data

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:

```python
>>> real = s.real  # real is a new HS signal accessing the same data
>>> s.real = new_real  # new_real can be an array or signal
>>> imag = s.imag  # imag is a new HS signal accessing the same data
>>> s.imag = new_imag  # 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:

```python
>>> 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.

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 angle() function. angle() will return an appropriate HyperSpy signal.

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 unwrap(), which uses the algorithm described in [Herraez].

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.

1.7 Signal1D Tools

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

1.7.1 Cropping

The crop_signal1D() crops 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.datasets.example_signals.EDS_TEM_Spectrum()
s.crop_signal1D(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 spectrum to the 5 keV-15 keV region:

s = hs.datasets.example_signals.EDS_TEM_Spectrum()
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.datasets.example_signals.EDS_TEM_Spectrum()
roi = hs.roi.SpanROI(left=5, right=15)
s.plot()
sc = roi.interactive(s)

### 1.7.2 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. Current background type supported are power law, offset, polynomial and gaussian. By default the background parameters are estimated using analytical approximations (keyword argument fast=True). For better accuracy, but higher processing time, the parameters can be estimated by curve fitting by setting fast=False.

Example of usage:

s = hs.datasets.artificial_data.get_core_loss_eels_signal(add_powerlaw=True)
s.remove_background(zero_fill=False)

### 1.7.3 Calibration

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

### 1.7.4 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()`

### 1.7.5 Integration

Deprecated since version 1.3: `integrate_in_range()`. It will be removed in 2.0. Use `integrate1D()` instead, possibly in combination with a Region Of Interest (ROI) if interactivity is required.
Fig. 13: Interactive spectrum cropping using a ROI.
Fig. 14: 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.
### 1.7.6 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()`

### 1.7.7 Spike removal

`spikes_removal_tool()` provides an user interface to remove spikes from spectra.

![Spikes removal tool](image)

**Fig. 15:** Spikes removal tool.

### 1.7.8 Peak finding

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

### 1.7.9 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()`
1.8 Signal2D Tools

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

1.8.1 Two dimensional signal registration (alignment)

New in version 1.4: sub_pixel_factor keyword.

The `align2D()` and `estimate_shift2D()` methods provide advanced image alignment functionality. Sub-pixel accuracy can be achieved by using skimage’s upsampled matrix-multiplication DFT method [Guizar2008]—by setting the `sub_pixel_factor` keyword argument—and/or, for multi-dimensional datasets only, using the statistical method [Schaffer2004]—by setting the `reference` keyword argument to "stat".

1.8.2 Cropping an image

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

```plaintext
>>> im = hs.datasets.example_signals.object_hologram()
>>> imc = im.crop(left=120, top=300, bottom=560) # im is cropped in-place
```

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

```plaintext
>>> im = hs.datasets.example_signals.object_hologram()
>>> # im is not cropped, imc is a "cropped view" of im
>>> imc = im.isig[120.:, 300.:560.]
```

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

```plaintext
>>> im = hs.datasets.example_signals.object_hologram()
>>> roi = hs.roi.RectangularROI(left=120, right=460., top=300, bottom=560)
>>> im.plot()
>>> imc = roi.interactive(im)
>>> imc.plot()
```

1.8.3 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.

1.9 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.

```plaintext
>>> s = hs.load('YourDataFilenameHere')
>>> s.plot()
```
Fig. 16: Interactive image cropping using a ROI.
if the object is single spectrum or an image one window will appear when calling the plot method.

### 1.9.1 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. 17: Visualisation of a 2D spectrum image.](image)

![Fig. 18: Visualisation of a 1D spectrum image.](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.

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 `hs.preferences.gui()`. Note that some combinations will not work for all platforms, as some systems reserve them for other purposes...

![Second Pointer](image.png)

**Fig. 19: 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>
<thead>
<tr>
<th>key</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>Switch second pointer on/off</td>
</tr>
<tr>
<td>Ctrl + Arrows</td>
<td>Change coordinates for dimensions 0 and 1 (typically x and y)</td>
</tr>
<tr>
<td>Shift + Arrows</td>
<td>Change coordinates for dimensions 2 and 3</td>
</tr>
<tr>
<td>Alt + Arrows</td>
<td>Change coordinates for dimensions 4 and 5</td>
</tr>
<tr>
<td>PageUp</td>
<td>Increase step size</td>
</tr>
<tr>
<td>PageDown</td>
<td>Decrease step size</td>
</tr>
<tr>
<td>+</td>
<td>Increase pointer size when the navigator is an image</td>
</tr>
<tr>
<td>-</td>
<td>Decrease pointer size when the navigator is an image</td>
</tr>
<tr>
<td>l</td>
<td>switch the scale of the y-axis between logarithmic and linear</td>
</tr>
</tbody>
</table>

To close all the figures run the following command:

```python
>>> 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').

1.9.2 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:

Fig. 20: Visualisation of a 1D image stack.

Fig. 21: Visualisation of a 2D image stack.

New in version 1.4: 1 keyboard shortcut

The following keyboard shortcuts are available when the 2D signal figure is in focus:
Table 4: Keyboard shortcuts available on the signal figure of 2D signal data

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

1.9.3 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.imshow arguments are supported as well:

```python
>>> import scipy

>>> img = hs.signals.Signal2D(scipy.misc.ascent())

>>> img.plot(colorbar=True, scalebar=False,
          axes_ticks=True, cmap='RdYlBu_r', saturated_pixels=0)
```

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 1.1.2: Passing keyword arguments to the navigator plot.

The same options can be passed to the navigator, albeit separately, by specifying them as a dictionary in `navigator_kwds` argument when plotting:

```python
>>> import numpy as np

>>> import scipy

>>> im = hs.signals.Signal2D(scipy.misc.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))
```

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:
Fig. 22: Custom colormap and switched off scalebar in an image.

Fig. 23: Custom different options for both signal and navigator image plots
>>> 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, cmap="RdBu", tight_layout=True)

Fig. 24: Divergent color map with Centre colormap enabled (default).

The same example with the feature disabled:

```python
>>> 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)
```

Fig. 25: Divergent color map with Centre colormap disabled.
1.9.4 Customizing the “navigator”

Data files used in the following examples can be downloaded using

```python
>>> #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",
... ".//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 P. Burdet, *et al.*, Acta Materialia, 61, p. 3090-3098 (2013) (see abstract).

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

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

![Fig. 26: Visualisation of a 3D image with a slider.](image)

**Fig. 26:** 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.

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

![Image](image1.png)

**Fig. 27:** Visualisation of a 3D spectrum image with sliders.

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.

```python
>>> 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)
```

![Image](image2.png)

**Fig. 28:** 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.

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

![Fig. 29: Visualisation of a stack of 2D spectral images.](image)

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.

```python
>>> 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. 30: Visualisation of a stack of 2D spectral images. The navigator is the “maximum spectrum”.](image)

Lastly, if no navigator is needed, “navigator=None” can be used.
1.9.5 Using Mayavi to visualize 3D data

Data files used in the following examples can be downloaded using

```python
>>> 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 `get_lines_intensity()` method can be found in *EDS lines intensity*.

```python
>>> 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. 31: Visualisation of isosurfaces with mayavi.](image)

Fig. 31: Visualisation of isosurfaces with mayavi.
1.9.6 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 `utils.plot` package.

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, 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):

```python
>>> import scipy

>>> image = hs.signals.Signal2D([scipy.misc.ascent()]*6)

>>> angles = hs.signals.BaseSignal(np.arange(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`:

```python
>>> import scipy

>>> image = hs.signals.Signal2D([scipy.misc.ascent()]*6)

>>> angles = hs.signals.BaseSignal(np.arange(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=[f'Rotation {i}$^\degree$'.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:

```python
>>> import scipy

>>> import numpy as np

>>> # load red channel of raccoon as an image

>>> image0 = hs.signals.Signal2D(scipy.misc.face()[:,:,0])
```
Fig. 32: Figure generated with `plot_images()` using the default values.

Turning Ascent

Rotation 10°  Rotation 20°  Rotation 30°

Rotation 40°  Rotation 50°  Rotation 60°

Fig. 33: Figure generated with `plot_images()` with customised labels.
>>> image0.metadata.General.title = 'Rocky Raccoon - R'
>>> # load ascent into a length 6 hyper-image
>>> image1 = hs.signals.Signal2D([scipy.misc.ascent()]*6)
>>> angles = hs.signals.BaseSignal(np.arange(10,70,10)).T
>>> image1.map(scipy.ndimage.rotate, angle=angles,
...            show_progressbar=False, 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.misc.face()[:,:,1])
>>> image2.metadata.General.title = 'Rocky Raccoon - G'
>>> # load rgb image of the raccoon
>>> rgb = hs.signals.Signal1D(scipy.misc.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, urlopen
>>> from zipfile import ZipFile
>>> files = urlretrieve("https://www.dropbox.com/s/ecd1gwjq04m5mx/"
...                     "HyperSpy_demos_EDX_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 used to call subplots_adjust method of matplotlib (see documentation).

>>> si_EDS = hs.load("core_shell.hdf5")
>>> im = si_EDS.get_lines_intensity()
>>> hs.plot.plot_images(im,
...    tight_layout=True, cmap='RdYlBu_r', axes_decor='off',
...    colorbar='single', saturated_pixels=2, 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})

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).
Fig. 34: Figure generated with `plot_images()` from a list of images.

X-ray line intensity of Core shell:

Fe_Ka at 6.40 keV

Pt_La at 9.44 keV

Fig. 35: Using `plot_images()` to plot the output of `get_lines_intensity()`.
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:

```python
>>> 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', saturated_pixels=2, scalebar=[0],
>>> scalebar_color='white', suptitle_fontsize=16)
```

Fig. 36: Using `plot_images()` to plot the output of `get_lines_intensity()` using a unique colormap for each image.

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.

```python
>>> 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
# 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)
```

(continues on next page)
>>> 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})

Fig. 37: 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.

**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.

### Plotting several spectra

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

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.
```python
>>> 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 = []

>>> for sigma in range(1, 10):
...     g.sigma.value = sigma
...     gs = m.as_signal()
...     gs.metadata.General.title = "sigma=%i" % sigma
...     gaussians.append(gs)
...     labels.append('sigma=%i' % sigma)

>>> hs.plot.plot_spectra(gaussians, legend='auto')
<matplotlib.axes.AxesSubplot object at 0x4c28c90>
```

![Figure generated by plot_spectra() using the overlap style.](image)

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:

```python
>>> import scipy.misc
>>> s = hs.signals.Signal1D(scipy.misc.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:

```python
>>> import scipy.misc
>>> s = hs.signals.Signal1D(scipy.misc.ascent()[100:160:10])
>>> color_list = ['red', 'red', 'blue', 'blue', 'red', 'red']
>>> line_style_list = ['-', '--', 'steps', '-', ':', '-']
>>> hs.plot.plot_spectra(s, style='cascade', color=color_list,
                          line_style=line_style_list, legend='auto')
```

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:

```python
>>> import scipy.misc
```

Fig. 39: Figure generated by `plot_spectra()` using the `cascade` style.

Fig. 40: Customising the line colors in `plot_spectra()`.
>>> fig, axarr = plt.subplots(1,2)
>>> s1 = hs.signals.Signal1D(scipy.misc.ascent()[100:160:10])
>>> s2 = hs.signals.Signal1D(scipy.misc.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)
>>> 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)
>>> axarr[0].set_xlabel('RdBu (colormap)')
>>> axarr[1].set_xlabel('summer (colormap)')
>>> fig.canvas.draw()

Fig. 41: Customising the line colors in `plot_spectra()` using a colormap.

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

```python
>>> import scipy.misc
>>> s = hs.signals.Signal1D(scipy.misc.ascent()[100:160:10])
>>> hs.plot.plot_spectra(s, style='heatmap')
```

```python
>>> import scipy.misc
>>> s = hs.signals.Signal1D(scipy.misc.ascent()[100:120:10])
```

(continues on next page)
Fig. 42: Figure generated by `plot_spectra()` using the `heatmap` style.

```python
>>> hs.plot.plot_spectra(s, style='mosaic')
```

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

```python
>>> import matplotlib.cm
>>> import scipy.misc

>>> s = hs.signals.Signal1D(scipy.misc.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:

```python
>>> import scipy.misc

>>> s = hs.signals.Signal1D(scipy.misc.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")
>>> cascade_plot.set_ylabel("Y-axis")
>>> cascade_plot.set_title("Cascade plot")
>>> plt.draw()
```

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

```python
>>> import scipy.misc

>>> s = hs.signals.Signal1D(scipy.misc.ascent()[100:160:10])
>>> cascade_plot = hs.plot.plot_spectra(s)
>>> cascade_plot.set_xlabel("An axis")
>>> cascade_plot.set_ylabel("Another axis")
>>> cascade_plot.set_title("A title!")
>>> plt.draw()
```

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:

```python
>>> import scipy.misc

>>> fig, axarr = plt.subplots(1,2)
```
Fig. 43: Figure generated by `plot_spectra()` using the *mosaic* style.

Fig. 44: Figure generated by `plot_spectra()` using the *heatmap* style showing how to customise the color map.
Fig. 45: Customising the figure with keyword arguments.

Fig. 46: Customising the figure by setting the matplotlib axes properties.
>>> s1 = hs.signals.Signal1D(scipy.misc.ascent()[100:160:10])
>>> s2 = hs.signals.Signal1D(scipy.misc.ascent()[200:260:10])
>>> hs.plot.plot_spectra(s1, style='cascade',
...                      color='blue', ax=axarr[0], fig=fig)
>>> hs.plot.plot_spectra(s2, style='cascade',
...                      color='red', ax=axarr[1], fig=fig)
>>> fig.canvas.draw()

Fig. 47: 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:

```python
>>> import scipy.misc
>>> s1 = hs.signals.Signal1D(scipy.misc.ascent()).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 navigator_options. To specify the navigator:

```python
>>> import scipy.misc
>>> s1 = hs.signals.Signal1D(scipy.misc.ascent()).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:

```python
>>> import scipy.misc
>>> s1 = hs.signals.Signal1D(scipy.misc.ascent()).as_signal1D(0).inav[:,:3]
>>> s2 = s1.deepcopy()*-1
```
Fig. 48: The `plot_signals()` plots several signals with optional synchronized navigation.

Fig. 49: Customising the navigator in `plot_signals()`.
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:

```python
>>> import scipy.misc
>>> s1 = hs.signals.Signal1D(scipy.misc.face()).as_signal1D(0)[...,3]
>>> s2 = s1.deepcopy()*-1
>>> hs.plot.plot_signals([s1, s2], sync=False, navigator_list=['slider', 'slider'])
```

![Fig. 50: Customising the navigator in plot_signals() by providing a navigator list.](image)

### 1.9.7 Markers

HyperSpy provides an easy access to the main marker of matplotlib. The markers can be used in a static way:

```python
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> m = hs.plot.markers.rectangle(x1=150, y1=100, x2=400, y2=400, color='red')
>>> 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.
Fig. 51: Disabling synchronized navigation in `plot_signals()`.

Fig. 52: Rectangle static marker.
>>> from skimage.feature import peak_local_max
>>> import scipy.misc

>>> ims = hs.signals.BaseSignal(scipy.misc.face()).as_signal2D([1,2])
>>> index = np.array([peak_local_max(im.data, min_distance=100,
...                   num_peaks=4)
...                     for im in ims])

>>> for i in range(4):
...     m = hs.plot.markers.point(x=index[:, i, 1],
...                     y=index[:, i, 0], color='red')
...     ims.add_marker(m)

Fig. 53: Point markers in image.

The 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.

>>> s = hs.signals.Signal1D(np.arange(100).reshape([10,10]))
>>> s.plot(navigator='spectrum')
>>> for i in range(s.axes_manager.shape[0]):
...     m = hs.plot.markers.text(y=s.sum(-1).data[i]+5,
...                     x=i, text='abcdefghij'[i])
...     s.add_marker(m, plot_on_signal=False)

>>> x = s.axes_manager.shape[-1]/2
...     m = hs.plot.markers.text(x=x, y=s.isig[x].data+2,
...                     text=[i for i in 'abcdefghij'])
...     s.add_marker(m)

Fig. 54: Multi-dimensional markers.

New in version 1.2: Permanent markers.

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

```python
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.markers.point(5, 9)
>>> s.add_marker(marker, permanent=True)

>>> s.plot()
```

Markers can be removed by deleting them from the metadata

```python
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.markers.point(5, 9)
>>> s.add_marker(marker, permanent=True)
>>> s.metadata.Markers.point = <marker.Point, point (x=5,y=9,color=black,size=20)>
>>> del s.metadata.Markers.point
>>> s.metadata.Markers # Returns nothing
```

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

```python
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.markers.point(5, 9)
```
Fig. 55: Plotting with permanent markers.

```python
>>> 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. For a signal with 1 navigation axis:

```python
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> marker = hs.markers.point((5, 1, 2), (9, 8, 1), color='red')
>>> s.add_marker(marker, permanent=True)
```

Fig. 56: Plotting with markers that change with the navigation index.

Or for a signal with 2 navigation axes:

```python
>>> s = hs.signals.Signal2D(np.arange(400).reshape(2, 2, 10, 10))
>>> marker = hs.markers.point(((5, 1), (1, 2)), ((2, 6), (9, 8)), color='red')
>>> s.add_marker(marker, permanent=True)
```

Fig. 57: Plotting with markers that change with the two-dimensional navigation index.

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

```python
>>> 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)
>>> marker = hs.markers.point(x=x, y=y, color='red')
>>> s.add_marker(marker, permanent=True)

New in version 1.2: markers keyword arguments can take an iterable in addition to single marker.

If you want to add a large amount of markers at the same time we advise to add them as an iterable (list, tuple, ...), which will be much faster:

```python
>>> from numpy.random import random

```
```python
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))

```
```python
>>> markers = (hs.markers.point(tuple(random() * 10 for i in range(3)),
...                              ...                              size=30, color=np.random.rand(3, 1))
...                              ...                              for i in range(500))

```
```python
>>> s.add_marker(markers, permanent=True)

```

Fig. 58: Plotting many markers with an iterable so they change with the navigation index.

This can also be done using different types of markers

```python
>>> from numpy.random import random

```
```python
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))

```
```python
>>> markers = []

```
```python
>>> for i in range(200):
...     markers.append(hs.markers.horizontal_line(
...                              ...                              tuple(random() * 10 for i in range(3)),
...                              ...                              color=np.random.rand(3, 1)))
...     markers.append(hs.markers.vertical_line(
...                              ...                              tuple(random() * 10 for i in range(3)),
...                              ...                              color=np.random.rand(3, 1)))
...     markers.append(hs.markers.point(
...                              ...                              tuple(random() * 10 for i in range(3)),
...                              ...                              color=np.random.rand(3, 1)))
...     markers.append(hs.markers.text(
...                              ...                              x=tuple(random() * 10 for i in range(3)),
...                              ...                              y=tuple(random() * 10 for i in range(3)),
...                              ...                              text=tuple("sometext" for i in range(3))))

```
```python
>>> s.add_marker(markers, permanent=True)

```

Fig. 59: 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:

```python
>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))

```
```python
>>> marker = hs.markers.point(2, 1, color='red')

```
```python
>>> s.add_marker(marker, plot_marker=False, permanent=True)

```
```python
>>> s.metadata.Markers

```
```python
point = <marker.Point, point (x=2,y=1,color=red,size=20)>

```
```python
>>> s.save("storing_marker.hdf5")

```
```python
>>> s1 = hs.load("storing_marker.hdf5")

```
```python
>>> s1.metadata.Markers

```
```python
point = <hyperspy.drawing._markers.point.Point object at 0x7efcfadb06d8>
```
1.10 Machine learning

1.10.1 Introduction

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.

The behaviour of some machine learning operations can be customised in the Machine Learning section Preferences.

Note: Currently the BSS algorithms operate on the result of a previous decomposition analysis. Therefore, it is necessary to perform a decomposition before attempting to perform a BSS.

1.10.2 Nomenclature

HyperSpy will decompose a dataset into two new datasets: one with the dimension of the signal space known as factors, and the other with the dimension of the navigation space known as loadings.

1.10.3 Decomposition

Decomposition techniques are most commonly applied as a means of noise reduction (or denoising) and dimensionality reduction.

Principal component analysis

One of the most popular decomposition methods is principal component analysis (PCA). To perform PCA on your dataset, run the decomposition() method:

```plaintext
>>> s.decomposition()
```

Note that the `s` variable must contain either a `BaseSignal` class or its subclasses, which will most likely have been loaded with the `load()` function, e.g. `s = hs.load('my_file.hspy')`. Also, the signal must be multi-dimensional, i.e. `s.axes_manager.navigation_size` must be greater than one.

Several algorithms exist for performing PCA, and the default algorithm in HyperSpy is SVD, which uses an approach called “singular value decomposition”. This method has many options, and for more information please read the method documentation.

Scree plots

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 and it should drop quickly, eventually becoming a slowly descending line.

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 below).

To obtain a scree plot for your dataset, run the `plot_explained_variance_ratio()` method:
>>> ax = s.plot_explained_variance_ratio(n=20)

Fig. 60: PCA scree plot

New in version 1.2.0: log, threshold, hline, 'vline', xaxis_type, xaxis_labeling, signal_fmt, noise_fmt, threshold, xaxis_type keyword arguments.

The default options for this method will plot a bare scree plot, but the method’s arguments allow for a great deal of customization. For example, 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). The number of significant components can be estimated and a vertical line drawn to represent this by specifying vline as True. In this case, the elbow or knee 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 selecting the point where this distance is largest. In the case of multiple occurrences of a maximum value 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 or knee finding technique can be found in [Satopää2011].

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.

Note that in the above figure, the first component has index 0. This is because Python uses zero based indexing i.e. the initial element of a sequence is found at index 0. To switch to a “number-based” (rather than “index-based”) notation, specify the xaxis_type parameter:

```python
>>> ax = s.plot_explained_variance_ratio(n=20,
... threshold=4,
... xaxis_type='number')
```
Fig. 61: PCA scree plot with number-based axis labeling and a threshold value specified.
Sometimes it can be useful to get the explained variance ratio as a spectrum, for example to plot several scree plots obtained using different data pre-treatments in the same figure using `plot_spectra()`. This can be achieved using `get_explained_variance_ratio()`.

**Denoising**

One of the most popular uses of PCA is data denoising. This is achieved by using a limited set of components to make a model of the original, omitting the later components that ideally contain only noise. This is also known as *dimensionality reduction*.

To perform this operation with HyperSpy, run the `get_decomposition_model()` method, usually after estimating the dimension of your data using a scree plot. For example:

```python
>>> sc = s.get_decomposition_model(components)
```

**Note:** The components argument can be one of several things (None, int, or list of ints):

- if None, all the components are used to construct the model.
- if int, only the given number of components (starting from index 0) are used to construct the model.
- if list of ints, only the components in the given list are used to construct the model.

Sometimes, it is useful to examine the residuals between your original data and the decomposition model. You can easily calculate and display the residuals:
>>> (s - sc).plot()

Hint: Unlike most of the analysis functions, this function returns a new object, which in the example above we have called `sc`. You can perform operations on this new object 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.

**Poissonian noise**

Many decomposition methods such as PCA assume that the noise of the data follows a Gaussian distribution. 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.

To perform Poissonian noise normalization:

```python
>>> # The long way:
>>> s.decomposition(normalize_poissonian_noise=True)

>>> # Because it is the first argument we could have simply written:
>>> s.decomposition(True)
```

More details about the scaling procedure can be found in [Keenan2004].

**Robust principal component analysis**

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 to consist of a low-rank component $L$ corrupted by a sparse error component $S$, then Robust PCA (RPCA) can be used to recover the low-rank component for subsequent processing [Candes2011].

The default RPCA algorithm is GoDec [Zhou2011]. In HyperSpy it returns the factors and loadings of $L$, and can be accessed with the following code. You must set the `output_dimension` when using RPCA.

```python
>>> s.decomposition(algorithm='RPCA_GoDec',
...     output_dimension=3)
```

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.

```python
>>> s.decomposition(algorithm='ORPCA',
...     output_dimension=3)
```

The online RPCA implementation sets several default parameters that are usually suitable for most datasets. However, to 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.

```python
>>> s.decomposition(algorithm='ORPCA',
...     output_dimension=3,
...     training_samples=32)
```

Finally, online RPCA includes three alternative methods to the default closed-form solver, which can again improve both the convergence and speed of the algorithm. These are particularly useful for very large datasets.

The first method is block-coordinate descent (BCD), and takes no additional parameters:
The second is based on stochastic gradient descent (SGD), and takes 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:

```python
>>> s.decomposition(algorithm='ORPCA',
...                   output_dimension=3,
...                   method='BCD')
```

The third method is MomentumSGD, which typically improves the convergence properties of stochastic gradient descent. This takes the further parameter “momentum”, which should be a fraction between 0 and 1.

```python
>>> s.decomposition(algorithm='ORPCA',
...                   output_dimension=3,
...                   method='SGD',
...                   learning_rate=1.1)
```

### Non-negative matrix factorization

Another popular decomposition method is non-negative matrix factorization (NMF), which can be accessed in HyperSpy with:

```python
>>> 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 [Nicolletti2013].

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 value for “output_dimension”.

### 1.10.4 Blind Source Separation

In some cases (it largely depends on the particular application) it is possible to obtain more physically interpretable set of components using a process called Blind Source Separation (BSS). For more information about blind source separation please see [Hyvarinen2000], and for an example application to EELS analysis, see [Pena2010].

To perform BSS on the result of a decomposition, run the `blind_source_separation()` method, e.g.:

```python
s.blind_source_separation(number_of_components)
```

**Note:** Currently the BSS algorithms operate on the result of a previous decomposition analysis. Therefore, it is necessary to perform a `Decomposition` first.
Note: You must pass an integer number of components to ICA. The best way to estimate this number in the case of a PCA decomposition is by inspecting the Scree plots.

### 1.10.5 Visualizing results

HyperSpy includes a number of plotting methods for the results of decomposition and blind source separation. All the methods begin with `plot_`:

1. `plot_decomposition_results()`.
2. `plot_decomposition_factors()`.
3. `plot_decomposition_loadings()`.
4. `plot_bss_results()`.
5. `plot_bss_factors()`.
6. `plot_bss_loadings()`.

1 and 4 (new in version 0.7) provide a more compact way of displaying the results. All the other methods display each component in its own window. For 2 and 3 it is wise to provide the number of factors or loadings you wish to visualise, since the default is to plot all of them. For BSS, the default is the number you included when running the `blind_source_separation()` method. In case of one dimensional factors or loadings, the latter can be toggled on and off by clicking on their corresponding line in the legend.

### 1.10.6 Obtaining 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()`.

### 1.10.7 Saving and loading results

There are several methods for storing the result of a machine learning analysis.

**Saving in the main file**

If you save the dataset on which you’ve performed machine learning analysis in the `HSpy - HyperSpy’s HDF5 Specification` format (the default in HyperSpy) (see Saving data to files), 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.
### Saving to an external file

Alternatively, you can save the results of the current machine learning analysis to a separate file with the `save()` method:

```python
Save the result of the analysis
>>> s.learning_results.save('my_results')

Load back the results
>>> s.learning_results.load('my_results.npz')
```

### Exporting in different formats

It is also possible to export the results of machine learning to any format supported by HyperSpy with:

- `export_decomposition_results()` or
- `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.

**Note:** Data exported in this way cannot be easily loaded into HyperSpy’s machine learning structure.

### 1.11 Model fitting

HyperSpy can perform curve fitting of one-dimensional signals (spectra) and two-dimensional signals (images) in n-dimensional data sets. Models can be created as a linear combination of predefined components and multiple optimisation algorithms can be used to fit the model to experimental data. Bounds and weights are supported. The syntax for creating both kinds of model is essentially the same, as in this documentation any method referred to in the `BaseModel` class is available for both kinds.

#### 2D models

Note that this first implementation lacks many of the features of 1D models e.g. plotting. Those will be added in future releases.

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`. At present plotting and gradient fitting methods tools for are not yet provided for the `Model2D` class.

#### 1.11.1 Binned/unbinned signals

Before creating a model verify that the `Signal.binned` metadata attribute of the signal is set to the correct value because the resulting model depends on this parameter. See Binned and unbinned signals for more details.

**Warning:** When importing data that have 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. We therefore strongly recommend that all pre-fitting binning should be done using Hyperspy.
1.11.2 Creating a model

A `Model1D` can be created for data in the `Signal1D` class using the `create_model()` method:

```python
>>> 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:

```python
>>> 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.

1.11.3 Creating components for the model

In HyperSpy a model consists of a linear combination of components and various components are available in one (components1d) and two-dimensions (components2d) to construct a model.

The following components are currently available for one-dimensional models:

- EELSCLEdge
- VolumePlasmonDrude
- PowerLaw
- Offset
- Exponential
- Expression
- ScalableFixedPattern
- Gaussian
- GaussianHF
- Lorentzian
- Voigt
- SkewNormal
- Polynomial
- Logistic
- Bleasdale
- Erf
- SEE
- Arctan
- HeavisideStep

The following components are currently available for two-dimensional models:

- Gaussian2D
• **Expression**

However, this doesn’t mean that you have to limit yourself to this meagre list of functions. A new function can easily be written as specified as below.

### Specifying custom components

New in version 1.2: **Expression** component can create 2D components.

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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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", )
```

Of course **Expression** is only useful for analytical functions. For more general components you need to create the component “by hand”. The good news is that, if you know how to write the function with Python, turning it into a component is very easy, just modify the following template to suit your needs:

```python
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'))
```

(continues on next page)
# Optionally we can set the initial values
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

If you need help with the task please submit your question to the users mailing list.

## 1.11.4 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:

```python
>>> m
<Model, title: my signal title>
>>> m.components # an empty model
<table>
<thead>
<tr>
<th>#</th>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
</table>
```
In fact, components may be created automatically in some cases. For example, if the `Signal1D` is recognised as EELS data, a power-law background component will automatically be placed in the model. To add a component, 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 for a type of data that can be modelled using Gaussians we might proceed as follows:

```python
>>> gaussian = hs.model.components1D.Gaussian()  # Create a Gaussian comp.
>>> m.append(gaussian)  # Add it to the model
>>> m.components  # Print the model components

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Gaussian</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>

```python
>>> gaussian2 = hs.model.components1D.Gaussian()  # Create another gaussian
>>> gaussian3 = hs.model.components1D.Gaussian()  # Create a third gaussian
```python
>>> m.extend((gaussian2, gaussian3))  # note the double parentheses!
```python
>>> m.components  # Print the model components

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Gaussian</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Gaussian_0</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian_1</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>

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.

```python
>>> m.extend([(gaussian2, gaussian3)])  # note the double parentheses!
```python
```python
>>> m.components  # Print the model components

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Carbon</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Long_Hydrogen_name</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Nitrogen</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>
```

We can customise the name of the components.

```python
>>> gaussian.name = 'Carbon'
>>> gaussian2.name = 'Long Hydrogen name'
>>> gaussian3.name = 'Nitrogen'
```python
```python
>>> m.components  # Print the model components

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Carbon</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Long_Hydrogen_name</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Nitrogen</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>
```

Two components cannot have the same name.

```python
>>> 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.

```python
>>> m

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Carbon</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Long_Hydrogen_name</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Nitrogen</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>
>>> m[0]
```

(continues on next page)
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.

```python
>>> 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 on simply set the `active` attribute back to `True`.

The current values of a component can be visualised using the `print_current_values()` method. The IPython display function elegantly presents it using HTML and allows for correct copying and pasting into Excel spreadsheets. Alternatively, a simpler form can be shown by setting the `fancy` argument to `False`.

```python
>>> m = s.create_model()
>>> m.fit()
>>> G = m[1]
>>> G.print_current_values(fancy=False)
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
```

In multidimensional 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`.

```python
>>> 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])
>>> g1.active_is_multidimensional = True
>>> g1._active_array
array([ True, True, True, True, True, True, True, True, True, True],
      dtype=bool)
>>> g2._active_array is None
True
>>> m.set_component_active_value(False)
>>> g1._active_array
array([False, False, False, False, False, False, False, False, False, False],
      dtype=bool)
>>> m.set_component_active_value(True, only_current=True)
>>> g1._active_array
array([ True, False, False, False, False, False, False, False, False, False],
      dtype=bool)
```
1.11.5 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 (Indexing). red_chisq and dof are automatically recomputed for the resulting slices.

```python
>>> 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)>
```

1.11.6 Getting and setting parameter values and attributes

`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. This can be changed to a standard terminal view with the argument `fancy=False`. 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 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()`. Example:

```python
>>> 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']
>>> g2.A.map['values']
>>> m.set_parameters_value('A', 40, only_current=True)
```
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:

```python
>>> g = hs.model.components1D.Gaussian()
>>> g.free_parameters
[<Parameter A of Gaussian component>,
 <Parameter sigma of Gaussian component>,
 <Parameter centre 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:

```python
>>> g1 = hs.model.components1D.Gaussian()
>>> g2 = hs.model.components1D.Gaussian()
>>> m.extend([g1,g2])
>>> m.set_parameters_not_free()
>>> g1.free_parameters
[]
>>> g2.free_parameters
[]
>>> m.set_parameters_free(parameter_name_list=['A'])
>>> g1.free_parameters
[<Parameter A of Gaussian component>]
>>> g2.free_parameters
[<Parameter A of Gaussian component>]
>>> m.set_parameters_free([g1], parameter_name_list=['sigma'])
>>> g1.free_parameters
[<Parameter A of Gaussian component>,
 <Parameter sigma of Gaussian component>]
>>> g2.free_parameters
[<Parameter A of Gaussian component>]
```

The value of a parameter can be coupled to the value of another by setting the twin attribute.

For example:

```python
>>> gaussian.parameters # Print the parameters of the gaussian components
(<Parameter A of Carbon component>,
 <Parameter sigma of Carbon component>,
 <Parameter centre of Carbon component>)
>>> gaussian.centre.free = False # Fix the centre
>>> gaussian.free_parameters # Print the free parameters
```

(continues on next page)
HyperSpy Documentation, Release 1.5.1.dev

>>> m.print_current_values(only_free=True, fancy=False)  # Print the values of all free parameters.

Model 1D:
Gaussian: Carbon
Active: True

Parameter Name | Free | Value | Std | Min | Max
=---------- | ----- | ---------- | ----- | ----- | -----
A | True | 1.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None

Gaussian: Long Hydrogen name
Active: True

Parameter Name | Free | Value | Std | Min | Max
=---------- | ----- | ---------- | ----- | ----- | -----
A | True | 1.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None
centre | True | 0.0 | None | None | None

Gaussian: Nitrogen
Active: True

Parameter Name | Free | Value | Std | Min | Max
=---------- | ----- | ---------- | ----- | ----- | -----
A | True | 1.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None
centre | True | 0.0 | None | None | None

>>> # 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(fancy=False)

Gaussian: Nitrogen
Active: True

Parameter Name | Free | Value | Std | Min | Max
=---------- | ----- | ---------- | ----- | ----- | -----
A | False | 5.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None
centre | True | 0.0 | None | None | None

Deprecation 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:

1.11. Model fitting
11.1.7 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()`.

The following table summarizes the features of the currently available optimizers. For more information on the local and global optimization algorithms, see the Scipy documentation.

New in version 1.1: Global optimizer Differential Evolution added.

Changed in version 1.1: `leastsq` supports bound constraints. `fmin_XXX` methods changed to the `scipy.optimize.minimize()` notation.

### Table 5: Features of curve fitting optimizers.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Bounds</th>
<th>Error estimation</th>
<th>Method</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>“leastsq”</td>
<td>Yes</td>
<td>Yes</td>
<td>‘ls’</td>
<td>local</td>
</tr>
<tr>
<td>“mpfit”</td>
<td>Yes</td>
<td>Yes</td>
<td>‘ls’</td>
<td>local</td>
</tr>
<tr>
<td>“odr”</td>
<td>No</td>
<td>Yes</td>
<td>‘ls’</td>
<td>local</td>
</tr>
<tr>
<td>“Nelder-Mead”</td>
<td>No</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“Powell”</td>
<td>No</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“CG”</td>
<td>No</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“BFGS”</td>
<td>No</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“Newton-CG”</td>
<td>No</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“L-BFGS-B”</td>
<td>Yes</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“TNC”</td>
<td>Yes</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>local</td>
</tr>
<tr>
<td>“Differential Evolution”</td>
<td>Yes</td>
<td>No</td>
<td>‘ls’, ‘ml’, ‘custom’</td>
<td>global</td>
</tr>
</tbody>
</table>

**Least squares with error estimation**

The following example shows how to perform least squares optimisation with error estimation.
First we create data consisting of a line \( y = ax + b \) with \( a = 1 \) and \( b = 100 \) and we add white noise to it:

```python
>>> s = hs.signals.Signal1D(
...   np.arange(100, 300))
>>> s.add_gaussian_noise(std=100)
```

To fit it we create a model consisting of a \textit{Polynomial} component of order 1 and fit it to the data.

```python
>>> m = s.create_model()
>>> line = hs.model.components1D.Polynomial(order=1)
>>> m.append(line)
>>> m.fit()
```

On fitting completion, the optimized value of the parameters and their estimated standard deviation are stored in the following line attributes:

```python
>>> line.a.value
0.9924615648843765
>>> line.b.value
103.67507406125888
>>> line.a.std
0.11771053738516088
>>> line.b.std
13.541061301257537
```

When the noise is heterocedastic, only if the \texttt{metadata.Signal.Noise\_properties.variance} attribute of the \texttt{Signal1D} instance is defined can the errors be estimated accurately. If the variance is not defined, the standard deviation of the parameters are still computed and stored in the \texttt{std} attribute by setting variance equal 1. However, the value won’t be correct unless an accurate value of the variance is defined in \texttt{metadata.Signal.Noise\_properties.variance}. See \textit{Setting the noise properties} for more information.

### Weighted least squares with error estimation

In the following example, we add poissonian noise to the data instead of gaussian noise and proceed to fit as in the previous example.

```python
>>> 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.coefficients.value
(1.0052331707848698, -1.0723588390873573)
>>> line.coefficients.std
(0.0081710549764721901, 1.4117294994070277)
```

Because the noise is heterocedastic, the least squares optimizer estimation is biased. A more accurate result can be obtained by using weighted least squares instead that, although still biased for poissonian noise, is a good approximation in most cases.

```python
>>> s.estimate_poissonian_noise_variance(
... expected_value=hs.signals.Signal1D(np.arange(300)))
>>> m.fit()
```
Maximum likelihood optimisation

We can use Poisson maximum likelihood estimation instead, which is an unbiased estimator for poissonian noise. To do so, we use a general optimizer called “Nelder-Mead”.

```python
>>> m.fit(fitter="Nelder-Mead", method="ml")
>>> line.coefficients.value
(1.0030718094185611, -0.63590210946134107)
```

Custom optimisations

New in version 1.4: Custom optimiser functions

Instead of the in-built least squares ('ls') and maximum likelihood ('ml') optimisation functions, a custom function can be passed to the model:

```python
>>> 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 optimiser (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
...     -------
...     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 np.sum(difference**2 * weights)
```
If the optimiser requires a gradient estimation function, it can be similarly passed, using the following signature:

```python
>>> 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
...         optimiser (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)
...     
...     # We use in-built jacobian estimation
...     jac = model._jacobian(values, data)
...     
...     return -(jac * (data / cur_value - 1)).sum(1)
```
>>> g1.centre.value = 7
>>> g1.centre.bmin = 7
>>> g1.centre.bmax = 14
>>> g1.centre.bounded = True
>>> m.fit(fitter="mpfit", bounded=True)
>>> m.print_current_values(fancy=False)

| Model1D: histogram                     |
| Gaussian: Gaussian                     |
| Active: True                           |

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Free</th>
<th>Value</th>
<th>Std</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>True</td>
<td>99997.3481</td>
<td>232.333693</td>
<td>0.0</td>
<td>None</td>
</tr>
<tr>
<td>sigma</td>
<td>True</td>
<td>0.00999184</td>
<td>2.68064163</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>centre</td>
<td>True</td>
<td>9.99980788</td>
<td>2.68064070</td>
<td>7.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>

### Goodness of fit

The chi-squared, reduced chi-squared and the degrees of freedom are computed automatically when fitting. They are stored as signals, in the `chisq`, `red_chisq` and `dof` attributes of the model respectively. Note that, unless `metadata.Signal.Noise_properties.variance` contains an accurate estimation of the variance of the data, the chi-squared and reduced chi-squared cannot be computed correctly. This is also true for homocedastic noise.

### Visualizing the model

To visualise the result use the `plot()` method:

```python
>>> 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()`:

```python
>>> m.plot(plot_components=True)  # Visualise the results
```

To disable this feature call `disable_plot_components()`.

New in version 1.4: `Signal1D.plot` keyword arguments

All extra keyword arguments are passes to the `plot()` method of the corresponding signal object. For example, the following plots the model signal figure but not its navigator:

```python
>>> 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()`.

### Setting the initial parameters

Non-linear regression 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 will be 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`. 
Fig. 63: 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.

**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()`
- `remove_signal_range()`
- `reset_signal_range()`

**Fitting multidimensional datasets**

To fit the model to all the elements of a multidimensional dataset use `multifit()`, e.g.:

```python
>>> m.multifit()  # warning: this can be a lengthy process on large datasets
```

`multifit()` fits the model at the first position, store 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...
Fig. 64: Interactive component position adjustment tool. Drag the vertical lines to set the initial value of the position parameter.
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.

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()`.

**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.

### 1.11.8 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`:

```python
>>> m = s.create_model()
>>> m.append(hs.model.components1D.Lorentzian())
>>> m.store('myname')
>>> s.models
    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
```

(continues on next page)
HyperSpy Documentation, Release 1.5.1.dev

(continued from previous page)

```python
>>> dimensions = (100)
>>> m1 = s.models.restore('myname')
>>> m1.components
```

<table>
<thead>
<tr>
<th>#</th>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Lorentzian</td>
<td>Lorentzian</td>
<td>Lorentzian</td>
</tr>
</tbody>
</table>

**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.

```python
>>> 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')  # or 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.

**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.

**1.11.9 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()`
1.11.10 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 will be described in full when accompanying paper is published, but we are making the implementation available now, with additional details available in the following conference proceeding.

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.

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 are 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.

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.

Global strategy family

Global strategies assume that the navigation coordinates of each pixel bear no relation to it’s 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 form 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.
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.

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:

```python
>>> 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 strategist list:

```bash
>>> 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:

```python
>>> samf.metadata.goodness_test.tolerance = 0.3 # use a sensible value
```

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:

```python
>>> samf.start(fitter='mpfit', bounded=True)
```
1.12 Electron Energy Loss Spectroscopy

1.12.1 Tools for EELS data analysis

The functions described in this chapter are only available for the `EELSSpectrum` class. To transform a `BaseSignal` (or subclass) into a `EELSSpectrum`:

```python
>>> s.set_signal_type("EELS")
```

Note these chapter discusses features that are available only for `EELSSpectrum` class. However, this class inherits many useful feature from its parent class that are documented in previous chapters.

Elemental composition of the sample

It can be useful to define the elemental composition of the sample for archiving purposes or to use some feature (e.g. curve fitting) that requires this information. The elemental composition of the sample can be declared using `add_elements()`. The information is stored in the `metadata` attribute (see `Metadata structure`). This information is saved to file when saving in the hspy format (HyperSpy's HDF5 specification).

Thickness estimation

The `estimate_thickness()` can estimate the thickness from a low-loss EELS spectrum using the Log-Ratio method.

Zero-loss peak centre and alignment

The `estimate_zero_loss_peak_centre()` can be used to estimate the position of the zero-loss peak. The method assumes that the ZLP is the most intense feature in the spectra. For a more general approach see `find_peaks1D_ohaver()`.

The `align_zero_loss_peak()` can align the ZLP with subpixel accuracy. It is more robust and easy to use than `align1D()` for the task. Note that it is possible to apply the same alignment to other spectra using the `also_align` argument. This can be useful e.g. to align core-loss spectra acquired quasi-simultaneously. If there are other features in the low loss signal which are more intense than the ZLP, the `signal_range` argument can narrow down the energy range for searching for the ZLP.

Deconvolutions

Three deconvolution methods are currently available:

- `fourier_log_deconvolution()`
- `fourier_ratio_deconvolution()`
- `richardson_lucy_deconvolution()`

Estimate elastic scattering intensity

The `estimate_elastic_scattering_intensity()` can be used to calculate the integral of the zero loss peak (elastic intensity) from EELS low-loss spectra containing the zero loss peak using the (rudimentary) threshold method. The threshold can be global or spectrum-wise. If no threshold is provided it is automatically calculated using `estimate_elastic_scattering_threshold()` with default values.
*estimate_elastic_scattering_threshold()* can be used to calculate separation point between elastic and inelastic scattering on EELS low-loss spectra. This algorithm calculates the derivative of the signal and assigns the inflexion point to the first point below a certain tolerance. This tolerance value can be set using the *tol* keyword. Currently, the method uses smoothing to reduce the impact of the noise in the measure. The number of points used for the smoothing window can be specified by the *npoints* keyword.

**Kramers-Kronig Analysis**

The single-scattering EEL spectrum is approximately related to the complex permittivity of the sample and can be estimated by Kramers-Kronig analysis. The *kramers_kronig_analysis()* method implements the Kramers-Kronig FFT method as in [Egerton2011] to estimate the complex dielectric function from a low-loss EELS spectrum. In addition, it can estimate the thickness if the refractive index is known and approximately correct for surface plasmon excitations in layers.

### 1.12.2 EELS curve fitting

HyperSpy makes it really easy to quantify EELS core-loss spectra by curve fitting as it is shown in the next example of quantification of a boron nitride EELS spectrum from the The EELS Data Base (see *Loading example data and data from online databases*).

Load the core-loss and low-loss spectra

```python
>>> s = hs.datasets.eelsdb(title="Hexagonal Boron Nitride",
... spectrum_type="coreloss")[0]
... >>> ll = hs.datasets.eelsdb(title="Hexagonal Boron Nitride",
... spectrum_type="lowloss")[0]
```

Set some important experimental information that is missing from the original core-loss file

```python
>>> s.set_microscope_parameters(beam_energy=100,
... convergence_angle=0.2,
... collection_angle=2.55)
```

**Warning:** *convergence_angle* and *collection_angle* are actually semi-angles and are given in mrad. *beam_energy* is in keV.

Define the chemical composition of the sample

```python
>>> s.add_elements(('B', 'N'))
```

In order to include the effect of plural scattering, the model is convolved with the loss loss spectrum in which case the low loss spectrum needs to be provided to *create_model()*:

```python
>>> m = s.create_model(ll=ll)
```

HyperSpy has created the model and configured it automatically:

```plaintext
# | Attribute Name | Component Name | Component Type
---- | -------------- | -------------- | --------------
0 | PowerLaw      | PowerLaw      | PowerLaw      
1 | N_K           | N_K           | EELSCLEdge    
2 | B_K           | B_K           | EELSCLEdge    
```
Conveniently, all the EELS core-loss components of the added elements are added automatically, names after its element symbol.

By default the fine structure features are disabled (although the default value can be configured (see Configuring HyperSpy)). We must enable them to accurately fit this spectrum.

```python
default = m.enable_fine_structure()
```

We use `smart_fit` instead of standard fit method because `smart_fit` is optimized to fit EELS core-loss spectra.

```python
>>> m.smart_fit()
```

This fit can also be applied over the entire signal to fit a whole spectrum image.

```python
>>> m.multifit(kind='smart')
```

Print the result of the fit:

```python
>>> m.quantify()
Absolute quantification:
Elem. Intensity
B  0.045648
N  0.048061
```

Visualize the result.

```python
>>> m.plot()
```

There are several methods that are only available in `EELSMoedl`:

- `smart_fit()` is a fit method that is more robust than the standard routine when fitting EELS data.
Quantify() prints the intensity at the current locations of all the EELS ionisation edges in the model.

remove_fine_structure_data() removes the fine structure spectral data range (as defined by the fine_structure_width) ionisation edge components. It is specially useful when fitting without convolving with a zero-loss peak.

The following methods permit to easily enable/disable background and ionisation edges components:

- enable_edges()
- enable_background()
- disable_background()
- enable_fine_structure()
- disable_fine_structure()

The following methods permit to easily enable/disable several ionisation edge functionalities:

- set_all_edges_intensities_positive()
- unset_all_edges_intensities_positive()
- enable_free_onset_energy()
- disable_free_onset_energy()
- fix_edges()
- free_edges()
- fix_fine_structure()
- free_fine_structure()

When fitting edges with fine structure enabled it is often desirable that the fine structure region of nearby ionization edges does not overlap. HyperSpy provides a method, resolve_fine_structure(), to automatically adjust the fine structure to prevent fine structure to avoid overlapping. This method is executed automatically when e.g. components are added or removed from the model, but sometimes is necessary to call it manually.

Sometimes it is desirable to disable the automatic adjustment of the fine structure width. It is possible to suspend this feature by calling suspend_auto_fine_structure_width(). To resume it use suspend_auto_fine_structure_width().

1.13 Energy-Dispersive X-ray Spectrometry (EDS)

The methods described in this chapter are specific to the following signals:

- EDSTEM
- EDSSEMSpectrum

This chapter describes step-by-step the analysis of an EDS spectrum (SEM or TEM).

Note: See also the EDS tutorials.
1.13.1 Spectrum loading and parameters

The sample and data used in this section are described in [Burdet2013], and can be downloaded using:

```python
>>> # 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",
... "/HyperSpy_demos_EDX_SEM_files.zip")
>>> with ZipFile("HyperSpy_demos_EDX_SEM_files.zip") as z:
...     z.extractall()
```

Loading data

All data are loaded with the `load()` function, as described in detail in *Loading files*. HyperSpy is able to import different formats, among them “.msa” and “.rpl” (the raw format of Oxford Instruments and Brucker).

Here are three example for files exported by Oxford Instruments software (INCA). For a single spectrum:

```python
>>> s = hs.load("Ni_superalloy_1pix.msa")
>>> s
<Spectrum1D, title: Signal1D, dimensions: (1024)>
```

For a spectrum image (The .rpl file is recorded as an image in this example, The method `as_signal1D()` set it back to a one dimensional signal with the energy axis in first position):

```python
>>> si = hs.load("Ni_superalloy_010.rpl").as_signal1D(0)
>>> si
<Spectrum1D, title: , dimensions: (256, 224, 1024)>
```

Finally, for a stack of spectrum images, using “*” as a wildcard character:

```python
>>> si4D = hs.load("Ni_superalloy_0*.rpl", stack=True)
>>> si4D
<Spectrum1D, title: , dimensions: (256, 224, 2, 1024)>
```

Microscope and detector parameters

First, the signal type (“EDS_TEM” or “EDS_SEM”) needs to be set with the `set_signal_type()` method. By assigning the class of the object, specific EDS methods are made available.

```python
>>> s = hs.load("Ni_superalloy_1pix.msa")
>>> s.set_signal_type("EDS_SEM")
>>> s
<EDSSpectrum, title: Signal1D, dimensions: (1024)>
```

You can also specify the signal type as an argument of the `load()` function:

```python
>>> s = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_SEM")
>>> s
<EDSSpectrum, title: Signal1D, dimensions: (1024)>
```

HyperSpy will automatically load any existing microscope parameters from the file, and store them in the metadata attribute (see *Metadata structure*). These parameters can be displayed as follows:
HyperSpy Documentation, Release 1.5.1.dev

>>> s = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_SEM")
>>> s.metadata.Acquisition_instrument.SEM
   Detector
     EDS
       azimuth_angle = 63.0
       elevation_angle = 35.0
       energy_resolution_MnKα = 130.0
       live_time = 0.006855
       real_time = 0.0
     beam_current = 0.0
     beam_energy = 15.0
     tilt_stage = 38.0

You can also set these parameters directly:

>>> s = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_SEM")
>>> s.metadata.Acquisition_instrument.SEM.beam_energy = 30

or by using the `set_microscope_parameters()` method:

>>> s = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_SEM")
>>> s.set_microscope_parameters(beam_energy = 30)

or through the GUI:

>>> s = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_SEM")
>>> s.set_microscope_parameters()

Fig. 65: EDS microscope parameters preferences window

Any microscope and detector parameters that are not found in the imported file will be set by default. These default values can be changed in the `Preferences` class (see `preferences`).

>>> hs.preferences.EDS.eds_detector_elevation = 37

or through the GUI:

>>> hs.preferences.gui()
Energy axis

The size, scale and units of the energy axis are automatically imported from the imported file, where they exist. These properties can also be set or adjusted manually with the AxesManager (see Axis properties for more info):

```python
>>> si = hs.load("Ni_superalloy_010.rpl",
...    signal_type="EDS_TEM").as_signal1D(0)
>>> si.axes_manager[-1].name = 'E'
>>> si.axes_manager['E'].units = 'keV'
>>> si.axes_manager['E'].scale = 0.01
>>> si.axes_manager['E'].offset = -0.1
```

or through the GUI:

```python
>>> si.axes_manager.gui()
```

Copying spectrum calibration

All of the above parameters can be copied from one spectrum to another with the get_calibration_from() method.

```python
>>> # slpixel contains all the parameters
>>> slpixel = hs.load("Ni_superalloy_1pix.msa", signal_type="EDS_TEM")
>>> # si contains no parameters
>>> si = hs.load("Ni_superalloy_010.rpl",
...    signal_type="EDS_TEM").as_signal1D(0)
>>> # Copy all the properties of slpixel to si
>>> si.get_calibration_from(slpixel)
```
1.13.2 Describing the sample

The description of the sample is also stored in the metadata attribute. It can be displayed using:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> s.metadata.Sample.thickness = 100
>>> s.metadata.Sample
    description = FePt bimetallic nanoparticles
    elements = ['Fe', 'Pt']
    thickness = 100
    xray_lines = ['Fe_Ka', 'Pt_La']
```

The following methods are either called “set” or “add”.

- “set” methods overwrite previously defined values
- “add” methods add to the previously defined values

Elements

The elements present in the sample can be defined using the set_elements() and add_elements() methods. Only element abbreviations are accepted:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.set_elements(['Fe', 'Pt'])
>>> s.add_elements(['Cu'])
>>> s.metadata.Sample
    elements = ['Cu', 'Fe', 'Pt']
```
X-ray lines

Similarly, the X-ray lines can be defined using the `set_lines()` and `add_lines()` methods. The corresponding elements will be added automatically. Several lines per element can be defined at once.

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.set_elements(['Fe', 'Pt'])
>>> s.set_lines(['Fe_Ka', 'Pt_La'])
>>> s.add_lines(['Fe_La'])
>>> s.metadata.Sample
    elements = ['Fe', 'Pt']
    xray_lines = ['Fe_Ka', 'Fe_La', 'Pt_La']
```

The X-ray lines can also be defined automatically, if the beam energy is set. The most excited X-ray line is selected per element (highest energy above an overvoltage of 2 (< beam energy / 2)):

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.set_elements(['Al', 'Cu', 'Mn'])
>>> s.set_microscope_parameters(beam_energy=30)
>>> s.add_lines()
>>> s.metadata.Sample
    elements = ['Al', 'Cu', 'Mn']
    xray_lines = ['Al_Ka', 'Cu_Ka', 'Mn_Ka']

>>> s.set_microscope_parameters(beam_energy=10)
>>> s.set_lines()
>>> s.metadata.Sample
    elements = ['Al', 'Cu', 'Mn']
    xray_lines = ['Al_Ka', 'Cu_La', 'Mn_La']
```

A warning is raised if you try to set an X-ray line higher than the beam energy:

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.set_elements(['Mn'])
>>> s.set_microscope_parameters(beam_energy=5)
>>> s.add_lines(['Mn_Ka'])
Warning: Mn Ka is above the data energy range.
```

Elemental database

HyperSpy includes an elemental database, which contains the energy of the X-ray lines.

```python
>>> hs.material.elements.Fe.General_properties
    Z = 26
    atomic_weight = 55.845
    name = iron

>>> hs.material.elements.Fe.Physical_properties
    density (g/cm^3) = 7.874

>>> hs.material.elements.Fe.Atomic_properties.Xray_lines
    Ka
        energy (keV) = 6.404
        weight = 1.0
    Kb
        energy (keV) = 7.0568
        weight = 0.1272
    La
```

(continues on next page)
Finding elements from energy

To find the nearest X-ray line for a given energy, use the utility function `get_xray_lines_near_energy()` to search the elemental database:

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> P = s.find_peaks1D_ohaver(maxpeakn=1)[0]
>>> hs.eds.get_xray_lines_near_energy(P['position'], only_lines=['a', 'b'])
['C_Ka', 'Ca_La', 'B_Ka']
```

The lines are returned in order of distance from the specified energy, and can be limited by additional, optional arguments.

Mass absorption coefficient database

A mass absorption coefficient database [Chantler2005] is available:

```python
>>> hs.material.mass_absorption_coefficient(
>>>     element='Al', energies=['C_Ka','Al_Ka'])
array([ 26330.38933818, 372.02616732])
```

```python
>>> hs.material.mass_absorption_mixture(
>>>     elements=['Al','Zn'], weight_percent=[50,50], energies='Al_Ka')
2587.4161643905127
```

1.13.3 Plotting

You can visualize an EDS spectrum using the `plot()` method (see `visualisation`). For example:

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot()
```

An example of multi-dimensional EDS data (e.g. 3D SEM-EDS) is given in `visualisation multi-dimension`.

Plotting X-ray lines

X-ray lines can be added as plot labels with `plot()`. The lines are either retrieved from `metadata.Sample.Xray_lines`, or selected with the same method as `add_lines()` using the elements in `metadata.Sample.elements`. 
Fig. 68: EDS spectrum

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.add_elements(['C', 'Mn', 'Cu', 'Al', 'Zr'])
>>> s.plot(True)
```

You can also select a subset of lines to label:

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.add_elements(['C', 'Mn', 'Cu', 'Al', 'Zr'])
>>> s.plot(True, only_lines=['Ka', 'b'])
```

### 1.13.4 Getting the intensity of an X-ray line

The sample and data used in this section are described in [Rossouw2015](#), and can be downloaded using:

```python
>>> # Download the data (1MB)
>>> from urllib.request import urlretrieve, urlopen
>>> from zipfile import ZipFile
>>> files = urlretrieve("https://www.dropbox.com/s/ecdlgwj04m5mx/"
... "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()
```

The width of integration is defined by extending the energy resolution of Mn Ka to the peak energy (`energy_resolution_MnKa` in the metadata):

```python
>>> s = hs.load('core_shell.hdf5')
>>> s.get_lines_intensity(['Fe_Ka'], plot_result=True)
```

The X-ray lines defined in `metadata.Sample.Xray_lines` are used by default. The EDS maps can be plotted using `plot_images()`, see [plotting several images](#) for more information in setting plotting parameters.
Fig. 69: EDS spectrum plot with line markers

Fig. 70: EDS spectrum plot with a selection of line markers
Fig. 71: Iron map as computed and displayed by `get_lines_intensity`.
HyperSpy Documentation, Release 1.5.1.dev

```python
>>> s = hs.load('core_shell.hdf5')
>>> s.metadata.Sample
   elements = ['Fe', 'Pt']
   xray_lines = ['Fe_Ka', 'Pt_La']
>>> eds_maps = s.get_lines_intensity()
>>> hs.plot.plot_images(eds_maps, axes_decor='off', scalebar='all')
```

X-ray line intensity of Core shell:

Fe\(_{Ka}\) at 6.40 keV  Pt\(_{La}\) at 9.44 keV

Finally, the windows of integration can be visualised using `plot()` method:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum().isig[5.:13.]
>>> s.add_lines()
>>> s.plot(integration_windows='auto')
```

Fig. 72: EDS spectrum with integration windows markers
Background subtraction

The background can be subtracted from the X-ray intensities with `get_lines_intensity()`. The background value is obtained by averaging the intensity in two windows on each side of the X-ray line. The position of the windows can be estimated using `estimate_background_windows()`, and can be plotted using `plot()`:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum().isig[5.:13.]
>>> s.add_lines()
>>> bw = s.estimate_background_windows(line_width=[5.0, 2.0])
>>> s.plot(background_windows=bw)
>>> s.get_lines_intensity(background_windows=bw, plot_result=True)
```

![Fig. 73: EDS spectrum with background subtraction markers.](image)

1.13.5 EDS Quantification

HyperSpy now includes three methods for EDS quantification:

- Cliff-Lorimer
- Zeta-factors
- Ionization cross sections

Quantification must be applied to the background-subtracted intensities, which can be found using `get_lines_intensity()`. The quantification of these intensities can then be calculated using `quantification()`.

The quantification method needs to be specified as either ‘CL’, ‘zeta’, or ‘cross_section’. If no method is specified, the function will raise an exception.

A list of factors or cross sections should be supplied in the same order as the listed intensities (please note that HyperSpy intensities in `get_lines_intensity()` are in alphabetical order).
A set of k-factors can be usually found in the EDS manufacturer software although determination from standard samples for the particular instrument used is usually preferable. In the case of zeta-factors and cross sections, these must be determined experimentally using standards.

Zeta-factors should be provided in units of kg/m^2. The method is described further in [Watanabe1996] and [Watanabe2006]. Cross sections should be provided in units of barns (b). Further details on the cross section method can be found in [MacArthur2016]. Conversion between zeta-factors and cross sections is possible using `edx_cross_section_to_zeta()` or `zeta_to_edx_cross_section()`.

Using the Cliff-Lorimer method as an example, quantification can be carried out as follows:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> kfactors = [1.450226, 5.075602]  # For Fe Ka and Pt La
>>> bw = s.estimate_background_windows(line_width=[5.0, 2.0])
>>> intensities = s.get_lines_intensity(background_windows=bw)
>>> atomic_percent = s.quantification(intensities, method='CL',
                                       factors=kfactors)
Fe (Fe_Ka): Composition = 15.41 atomic percent
Pt (Pt_La): Composition = 84.59 atomic percent
```

The obtained composition is in atomic percent, by default. However, it can be transformed into weight percent either with the option `quantification()`:

```python
>>> # With s, intensities and kfactors from before
>>> s.quantification(intensities, method='CL', factors=kfactors,
                     composition_units='weight')
Fe (Fe_Ka): Composition = 4.96 weight percent
Pt (Pt_La): Composition = 95.04 weight percent
```

or using `atomic_to_weight()`:

```python
>>> # With atomic_percent from before
>>> weight_percent = hs.material.atomic_to_weight(atomic_percent)
```

The reverse method is `weight_to_atomic()`.

The zeta-factor method needs both the ‘beam_current’ (in nA) and the acquisition or dwell time (referred to as ‘real_time’ in seconds) in order to obtain an accurate quantification. Both of these parameters can be assigned to the metadata using:

```python
>>> s.set_microscope_parameters(beam_current=0.5)
>>> s.set_microscope_parameters(real_time=1.5)
```

If these parameters are not set, the code will produce an error. The zeta-factor method will produce two sets of results. Index [0] contains the composition maps for each element in atomic percent, and index [1] contains the mass-thickness map.

The cross section method needs the ‘beam_current’, dwell time (‘real_time’) and probe area in order to obtain an accurate quantification. The ‘beam_current’ and ‘real_time’ can be set as shown above. The ‘probe_area’ (in nm^2) can be defined in two different ways.

If the probe diameter is narrower than the pixel width, then the probe is being under-sampled and an estimation of the probe area needs to be used. This can be added to the metadata with:

Alternatively, if sub-pixel scanning is used (or the spectrum map was recorded at a high spatial sampling and subsequently binned into much larger pixels) then the illumination area becomes the pixel area of the spectrum image. This is a much more accurate approach for quantitative EDS and should be used where possible. The pixel width could
either be added to the metadata by putting the pixel area in as the ‘probe_area’ (above) or by calibrating the spectrum image (see Setting axis properties).

Either approach will provide an illumination area for the cross_section quantification. If the pixel width is not set, the code will still run with the default value of 1 nm with a warning message to remind the user that this is the case.

The cross section method will produce two sets of results. Index [0] contains the composition maps for each element in atomic percent and index [1] is the number of atoms per pixel for each element.

**Note:** Please note that the function does not assume square pixels, so both the x and y pixel dimensions must be set. For quantification of line scans, rather than spectrum images, the pixel area should be added to the metadata as above.

### 1.13.6 EDS curve fitting

The intensity of X-ray lines can be extracted using curve-fitting in HyperSpy. This example uses an EDS-SEM spectrum of a a test material (EDS-TM001) provided by BAM.

First, we load the spectrum, define the chemical composition of the sample and set the beam energy:

```python
>>> s = hs.load('bam.msa')
>>> s.add_elements(['Al', 'Ar', 'C', 'Cu', 'Mn', 'Zr'])
>>> s.set_microscope_parameters(beam_energy=10)
```

Next, the model is created with `create_model()`. One Gaussian is automatically created per X-ray line, along with a polynomial for the background.

```python
>>> m = s.create_model()
>>> m.print_current_values()

<table>
<thead>
<tr>
<th>Components</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al_Ka</td>
<td>A</td>
<td>65241.4</td>
</tr>
<tr>
<td>Al_Kb</td>
<td>A</td>
<td>3136.88</td>
</tr>
<tr>
<td>Ar_Ka</td>
<td>A</td>
<td>79258.9</td>
</tr>
<tr>
<td>Ar_Kb</td>
<td>A</td>
<td>1640.8</td>
</tr>
<tr>
<td>C_Ka</td>
<td>A</td>
<td>74032.6</td>
</tr>
<tr>
<td>Cu_Ka</td>
<td>A</td>
<td>47796.6</td>
</tr>
<tr>
<td>Cu_Kb</td>
<td>A</td>
<td>73665.7</td>
</tr>
<tr>
<td>Cu_La</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Cu_Lb1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Cu_Ln</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Cu_L1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Cu_Lb3</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_Ka</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_Kb</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_La</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_Ln</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_L1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Mn_Lb3</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
```

(continues on next page)
The width and the energies are fixed, while the heights of the sub-X-ray lines are linked to the main X-ray lines (alpha lines). The model can now be fitted:

```python
>>> m.fit()
```

The background fitting can be improved with `fit_background()` by enabling only energy ranges containing no X-ray lines:

```python
>>> m.fit_background()
```

The width of the X-ray lines is defined from the energy resolution (FWHM at Mn Ka) provided by `energy_resolution_MnKa` in `metadata`. This parameter can be calibrated by fitting with `calibrate_energy_axis()`:

```python
>>> m.calibrate_energy_axis(calibrate='resolution')
```

Energy resolution (FWHM at Mn Ka) changed from 130.000000 to 131.927922 eV

Fine-tuning of specific X-ray lines can be achieved using `calibrate_xray_lines()`:

```python
>>> m.calibrate_xray_lines('energy', ['Ar_Ka'], bound=10)
>>> m.calibrate_xray_lines('width', ['Ar_Ka'], bound=10)
>>> m.calibrate_xray_lines('sub_weight', ['Mn_La'], bound=10)
```

The result of the fit is obtained with the `get_lines_intensity()` method:

```python
>>> result = m.get_lines_intensity(plot_result=True)
```

The following methods can be used to enable/disable different functionalities of X-ray lines when fitting:

- `free_background()`
- `fix_background()`
- `enable_xray_lines()`
• `disable_xray_lines()`
• `free_sub_xray_lines_weight()`
• `fix_sub_xray_lines_weight()`
• `free_xray_lines_energy()`
• `fix_xray_lines_energy()`
• `free_xray_lines_width()`
• `fix_xray_lines_width()`

1.14 Dielectric function tools

The `DielectricFunction` class inherits from `ComplexSignal` and can thus access complex properties. To convert a `ComplexSignal` to a `DielectricFunction`, make sure that the signal dimension and signal type are properly set:

```python
>>> s.set_signal_type('DielectricFunction')
```

Note that `DielectricFunction` is complex and therefore is a subclass of `ComplexSignal1D`. 

1.14. Dielectric function tools 147
1.14.1 Number of effective electrons

The Bethe f-sum rule gives rise to two definitions of the effective number (see [Egerton2011]):

\[
n_{\text{eff}1} \left( -3 \left( \epsilon^{-1} \right) \right) = \frac{2\epsilon_0 m_0}{\pi \hbar^2 e^2 n_a} \int_0^E E' \left( \frac{-1}{\epsilon} \right) dE' \\
n_{\text{eff}2} (\epsilon_2) = \frac{2\epsilon_0 m_0}{\pi \hbar^2 e^2 n_a} \int_0^E E' \epsilon_2 (E') dE'
\]

where \( n_a \) is the number of atoms (or molecules) per unit volume of the sample, \( \epsilon_0 \) is the vacuum permittivity, \( m_0 \) is the electron mass and \( e \) is the electron charge.

The `get_number_of_effective_electrons()` method computes both.

1.14.2 Compute the electron energy-loss signal

The `get_electron_energy_loss_spectrum()` “naively” computes the single-scattering electron-energy loss spectrum from the dielectric function given the zero-loss peak (or its integral) and the sample thickness using:

\[
S(E) = \frac{2 I_0 t}{\pi a_0 m_0 v^2} \ln \left[ 1 + \left( \frac{\beta}{\theta(E)} \right)^2 \right] \Im \left[ \frac{-1}{\epsilon(E)} \right]
\]

where \( I_0 \) is the zero-loss peak integral, \( t \) the sample thickness, \( \beta \) the collection semi-angle and \( \theta(E) \) the characteristic scattering angle.

1.15 Electron Holography

HyperSpy provides the user with a signal class which can be used to process electron holography data:

- **HologramImage**

It inherits from `Signal2D` class and thus can use all of its functionality. The usage of the class is explained in the following sections.

1.15.1 The HologramImage class

The `HologramImage` class is designed to contain images acquired via electron holography.

To transform a `Signal2D` (or subclass) into a `HologramImage` use:

```python
>>> im.set_signal_type('hologram')
```

Reconstruction of holograms

The detailed description of electron holography and reconstruction of holograms can be found in literature [Gabor1948], [Tonomura1999], [McCartney2007], and [Joy1993]. Fourier based reconstruction of off-axis holograms (includes finding a side band in FFT, isolating and filtering it, recenter and calculate inverse Fourier transform) can be performed using the `reconstruct_phase()` method which returns a `Complex2D` class, containing the reconstructed electron wave. The `reconstruct_phase()` method takes sideband position and size as parameters:
```python
>>> import hyperspy.api as hs
>>> im = hs.datasets.example_signals.object_hologram()
>>> wave_image = im.reconstruct_phase(sb_position=(<y>, <x>),
                               sb_size=sb_radius)
```

The parameters can be found automatically by calling following methods:

```python
>>> sb_position = im.estimate_sideband_position(ap_cb_radius=None,
                                           sb='lower')
>>> sb_size = im.estimate_sideband_size(sb_position)
```

*estimate_sideband_position()* method searches for maximum of intensity in upper or lower part of FFT pattern (parameter `sb`) excluding the middle area defined by `ap_cb_radius`. `estimate_sideband_size()` method calculates the radius of the sideband filter as half of the distance to the central band which is commonly used for strong phase objects. Alternatively, the sideband filter radius can be recalculate as 1/3 of the distance (often used for weak phase objects) for example:

```python
>>> sb_size = sb_size * 2 / 3
```

To reconstruct the hologram with a vacuum reference wave, the reference hologram should be provided to the method either as Hyperspy’s *HologramImage* or as a nparray:

```python
>>> reference_hologram = hs.datasets.example_signals.reference_hologram()
>>> wave_image = im.reconstruct_phase(reference_hologram,
                      sb_position=sb_position,
                      sb_size=sb_sb_size)
```

Using the reconstructed wave, one can access its amplitude and phase (also unwrapped phase) using `amplitude` and `phase` properties (also the `unwrapped_phase()` method):

```python
>>> wave_image.unwrapped_phase().plot()
```

Additionally, it is possible to change the smoothness of the sideband filter edge (which is by default set to 5% of the filter radius) using parameter `sb_smoothness`.

Both `sb_size` and `sb_smoothness` can be provided in desired units rather than pixels (by default) by setting `sb_unit` value either to `mrad` or `nm` for milliradians or inverse nanometers respectively. For example:

```python
>>> wave_image = im.reconstruct_phase(reference_hologram,
                      sb_position=sb_position, sb_size=30,
                      sb_smoothness=0.05*30, sb_unit='mrad')
```

Also the `reconstruct_phase()` method can output wave images with desired size (shape). By default the shape of the original hologram is preserved. Though this leads to oversampling of the output wave images, since the information is limited by the size of the sideband filter. To avoid oversampling the output shape can be set to the diameter of the sideband as follows:

```python
>>> wave_image = im.reconstruct_phase(reference_hologram,
                      sb_position=sb_position,
                      sb_size=sb_sb_size,
                      output_shape=(2*sb_size, 2*sb_size))
```

Note that the `reconstruct_phase()` method can be called without parameters, which will cause their automatic assignment by `estimate_sideband_position()` and `estimate_sideband_size()` methods. This, however, is not recommended for not experienced users.
Fig. 74: Unwrapped phase image.
Getting hologram statistics

There are many reasons to have an access to some parameters of holograms which describe the quality of the data. `statistics()` can be used to calculate carrier frequency, fringe spacing and estimate fringe contrast. The method outputs dictionary with the values listed above calculated also in different units. In particular fringe spacing is calculated in pixels (fringe sampling) as well as in calibrated units. Carrier frequency is calculated in inverse pixels or calibrated units as well as radians. Estimation of fringe contrast is either performed by division of standard deviation by mean value of hologram or in Fourier space as twice the fraction of amplitude of sideband centre and amplitude of center band (i.e. FFT origin). The first method is default and using it requires the fringe field to cover entire field of view; the method is highly sensitive to any artifacts in holograms like dud pixels, fresnel fringes and etc. The second method is less sensitive to the artifacts listed above and gives reasonable estimation of fringe contrast even if the hologram is not covering entire field of view, but it is highly sensitive to precise calculation of sideband position and therefore sometimes may underestimate the contrast. The selection between to algorithms can be done using parameter `fringe_contrast_algorithm` setting it to 'statistical' or to 'fourier'. The side band position typically provided by a `sb_position`. The statistics can be accessed as follows:

```python
>>> statistics = im.statistics(sb_position=sb_position)
```

Note that by default the `single_value` parameter is `True` which forces the output of single values for each entry of statistics dictionary calculated from first navigation pixel. (I.e. for image stacks only first image will be used for calculating the statistics.) Otherwise:

```python
>>> statistics = im.statistics(sb_position=sb_position, single_value=False)
```

Entries of `statistics` are Hyperspy signals containing the hologram parameters for each image in a stack.

The estimation of fringe spacing using 'fourier' method applies apodization in real space prior calculating FFT. By default `apodization` parameter is set to `hanning` which applies Hanning window. Other options are using either `None` or `hamming` for no apodization or Hamming window. Please note that for experimental conditions especially with extreme sampling of fringes and strong contrast variation due to Fresnel effects the calculated fringe contrast provides only an estimate and the values may differ strongly depending on apodization.

For further information see documentation of `statistics()`.

### 1.16 Loading and saving data

**Contents**

- Loading and saving data
  - Loading files: the `load` function
    - Loading multiple files
  - Saving data to files
  - Supported formats
    - **HyperSpy - HyperSpy’s HDF5 Specification**
      - Extra saving arguments
    - **NetCDF**
    - **MRC**
    - **MRCZ**
HyperSpy can read and write to multiple formats (see Supported formats). To load data use the `load()` command. For example, to load the image ascent.jpg you can type:

```python
>>> s = hs.load("ascent.jpg")
```

If the loading was successful, the variable `s` contains a generic `BaseSignal`, a `Signal1D` or an `Signal2D`. 

### 1.16.1 Loading files: the load function

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

```python
>>> s = hs.load("ascent.jpg")
```

If the loading was successful, the variable `s` contains a generic `BaseSignal`, a `Signal1D` or an `Signal2D`. 

152 Chapter 1. HyperSpy User Guide
HyperSpy will try to guess the most likely data type for the corresponding file. However, you can force it to read the data as a particular data type by providing the signal keyword, which has to be one of: spectrum, image or EELS, e.g.:

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

Some file formats store some extra information about the data, which can be stored in “attributes”. If HyperSpy manages to read some extra information about the data it stores it in the original_metadata attribute. Also, it is possible that other information will be mapped by HyperSpy to a standard location where it can be used by some standard routines, the metadata attribute.

To print the content of the parameters simply:

```python
>>> s.metadata
```

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

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

Deprecated since version 1.2: memmap_dir and load_to_memory load() keyword arguments. Use lazy instead of load_to_memory. lazy makes memmap_dir unnecessary.

Almost all file readers support accessing the data without reading it to memory (see Supported formats for a list). This feature can be useful when analysing large files. To load a file without loading it to memory simply set lazy to True e.g.:

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

More details on lazy evaluation support in Working with big data.

**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.:

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

or by using shell-style wildcards

New in version 1.2.0: stack multi-signal files

By default HyperSpy will return a list of all the files loaded. Alternatively, HyperSpy can stack the data of 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 numbers of signals per file must also match, or an error will be raised.

It is also possible to load multiple files with a single command without stacking them by passing the stack=False argument to the load function, in which case the function will return a list of objects, e.g.:
1.16.2 Saving data to files

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 - HyperSpy's HDF5 Specification`) 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 - HyperSpy's HDF5 Specification` format:

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

If instead you want to save in the Ripple write instead:

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

Some formats take extra arguments. See the relevant subsection of `Supported formats` for more information.

1.16.3 Supported formats

Here is a summary of the different formats that are currently supported by HyperSpy. The “lazy” column specifies if lazy evaluation is supported.
### Table 6: Supported file formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Read</th>
<th>Write</th>
<th>lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gatan’s dm3</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Gatan’s dm4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FEI’s emi and ser</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HDF5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Image: jpg</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TIFF</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MRC</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MRCZ</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EMD/MSA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NetCDF</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ripple</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SEMPER unf</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Blockfile</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DENS heater log</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bruker’s bcf</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Bruker’s spx</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>EMD (NCEM)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EMD (Velox)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Protochips log</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>EDAX .spc and .spd</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### HSpy - HyperSpy’s HDF5 Specification

This is the default format and it is the only one that guarantees that no information will be lost in the writing process and that supports saving data of arbitrary dimensions. It is based on the **HDF5 open standard**. The HDF5 file format is supported by many applications. Part of the specification is documented in **Metadata structure**.

New in version 1.2: Enable saving HSpy files with the `.hspy` extension. Previously only the `.hdf5` extension was recognised.

Changed in version 1.3: The default extension for the HyperSpy HDF5 specification is now `.hspy`. The option to change the default is no longer present in **preferences**.

Only loading of HDF5 files following the HyperSpy specification are supported. Usually their extension is `.hspy` extension, but older versions of HyperSpy would save them with the `.hdf5` extension. Both extensions are recognised by HyperSpy since version 1.2. However, HyperSpy versions older than 1.2 won’t recognise the `.hspy` extension. To workaround the issue when using old HyperSpy installations simply change the extension manually to `.hdf5` or save directly the file using this extension by explicitly adding it to the filename e.g.:

```python
>>> s = hs.signals.BaseSignal([0])
>>> s.save('test.hdf5')
```

When saving to **hspy**, all supported objects in the signal's **metadata** is stored. This includes lists, tuples and signals. Please note that in order to increase saving efficiency and speed, if possible, the inner-most structures are converted to numpy arrays when saved. This procedure homogenizes any types of the objects inside, most notably casting numbers as strings if any other strings are present:

```python
>>> # before saving:
>>> somelist
[1, 2.0, 'a name']
```

(continues on next page)
# after saving:
['1', '2.0', 'a name']

The change of type is done using numpy “safe” rules, so no information is lost, as numbers are represented to full machine precision.

This feature is particularly useful when using get_lines_intensity() (see get lines intensity):

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.metadata.Sample.intensities = s.get_lines_intensity()
>>> s.save('EDS_spectrum.hspy')

>>> s_new = hs.load('EDS_spectrum.hspy')
>>> s_new.metadata.Sample.intensities
[<BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Al_Ka at 1.49 keV>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: C_Ka at 0.28 keV>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Cu_La at 0.93 keV>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Mn_La at 0.63 keV>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Zr_La at 2.04 keV>]
```

New in version 1.3.1: chunks keyword argument

By default, the data is saved in chunks that are optimised to contain at least one full signal. It is possible to customise the chunk shape using the chunks keyword. For example, to save the data with (20, 20, 256) chunks instead of the default (7, 7, 2048) chunks for this signal:

```python
>>> s = hs.signals.Signal1D(np.random.random((100, 100, 2048)))
>>> s.save("test_chunks", chunks=(20, 20, 256), overwrite=True)
```

Note that currently it is not possible to pass different customised chunk shapes to all signals and arrays contained in a signal and its metadata. Therefore, the value of chunks provided on saving will be applied to all arrays contained in the signal.

By passing True to chunks the chunk shape is guessed using h5py’s guess_chunks function what, for large signal spaces usually leads to smaller chunks as guess_chunks does not impose the constrain of storing at least one signal per chunks. For example, for the signal in the example above passing chunks=True results in (7, 7, 256) chunks.

**Extra saving arguments**

- **compression**: One of None, ‘gzip’, ‘szip’, ‘lzf’ (default is ‘gzip’).

**NetCDF**

This was the default format in HyperSpy’s predecessor, EELSLab, but it has been superseded by HDF5 in HyperSpy. We provide only reading capabilities but we do not support writing to this format.

Note that only NetCDF files written by EELSLab are supported.

To use this format a python netcdf interface must be installed manually because it is not installed by default when using the automatic installers.
MRC

This is a format widely used for tomographic data. Our implementation is based on this specification. We also partly support FEI’s custom header. We do not provide writing features for this format, but, as it is an open format, we may implement this feature in the future on demand.

For mrc files load takes the mmap_mode keyword argument enabling loading the file using a different mode (default is copy-on-write). However, note that lazy loading does not support in-place writing (i.e lazy loading and the “r+” mode are incompatible).

MRCZ

MRCZ is an extension of the CCP-EM MRC2014 file format. It uses the blosc meta-compression library to bitshuffle and compress files in a blocked, multi-threaded environment. The supported data types are:

\[ \text{float32, int8, uint16, int16, complex64} \]

It supports arbitrary meta-data, which is serialized into JSON.

MRCZ also supports asynchronous reads and writes.

Repository: https://github.com/em-MRCZ  PyPI: https://pypi.python.org/pypi/mrcz  Citation: Submitted. Preprint: http://www.biorxiv.org/content/early/2017/03/13/116533

Support for this format is not enabled by default. In order to enable it install the mrcz and optionally the blosc Python packages.

Extra saving arguments

do_async: currently supported within Hyperspy for writing only, this will save the file in a background thread and return immediately. Defaults to False.

Warning: There is no method currently implemented within Hyperspy to tell if an asynchronous write has finished.

compressor: The compression codec, one of [None, ‘zlib’, ‘zstd’, ‘lz4’]. Defaults to None.

clevel: The compression level, an int from 1 to 9. Defaults to 1.

n_threads: The number of threads to use for blosc compression. Defaults to the maximum number of virtual cores (including Intel Hyperthreading) on your system, which is recommended for best performance. If do_asyc = True you may wish to leave one thread free for the Python GIL.

The recommended compression codec is ‘zstd’ (zStandard) with clevel=1 for general use. If speed is critical, use ‘lz4’ (LZ4) with clevel=9. Integer data compresses more reliably than floating-point data, and in general the histogram of values in the data reflects how compressible it is.

To save files that are compatible with other programs that can use MRC such as GMS, IMOD, Relion, MotionCorr, etc. save with compressor=None, extension .mrc. JSON metadata will not be recognized by other MRC-supporting software but should not cause crashes.

Example Usage
HyperSpy is able to read and write data to all the image formats supported by the Python Image Library (PIL). This includes png, pdf, gif etc. It is important to note that these image formats only support 8-bit files, and therefore have an insufficient dynamic range for most scientific applications. It is therefore highly discouraged to use any general image format (with the exception of **TIFF**) which uses another library to store data for analysis purposes.
TIFF

HyperSpy can read and write 2D and 3D TIFF files using using Christoph Gohlke’s tifffile library. In particular it supports reading and writing of TIFF, BigTIFF, OME-TIFF, STK, LSM, NIH, and FluoView files. Most of these are uncompressed or losslessly compressed 2**(0 to 6) bit integer,16, 32 and 64-bit float, grayscale and RGB(A) images, which are commonly used in bio-scientific imaging. See the library webpage for more details.

Currently HyperSpy has limited support for reading and saving the TIFF tags. However, the way that HyperSpy reads and saves the scale and the units of tiff files is compatible with ImageJ/Fiji and Gatan Digital Micrograph software. HyperSpy can also import the scale and the units from tiff files saved using FEI and Zeiss SEM software.

```python
>>> # Force read image resolution using the x_resolution, y_resolution and
>>> # the resolution_unit of the tiff tags. Be aware, that most of the
>>> # software doesn't (properly) use these tags when saving tiff files.
>>> s = hs.load('file.tif', force_read_resolution=True)
```

HyperSpy can also read and save custom tags through Christoph Gohlke’s tifffile library. See the library webpage for more details.

```python
>>> # Saving the string 'Random metadata' in a custom tag (ID 65000)
>>> extratag = [(65000, 's', 1, "Random metadata", False)]
>>> s.save('file.tif', extratags=extratag)

>>> # Saving the string 'Random metadata' from a custom tag (ID 65000)
>>> s2 = hs.load('file.tif')
>>> s2.original_metadata['Number_65000']
'b'Random metadata'
```

Gatan Digital Micrograph

HyperSpy can read both dm3 and dm4 files but the reading features are not complete (and probably they will be unless Gatan releases the specifications of the format). That said, we understand that this is an important feature and if loading a particular Digital Micrograph file fails for you, please report it as an issue in the issues tracker to make us aware of the problem.

Extra loading arguments

- `optimize`: bool, default is True. During loading, the data is replaced by its optimized copy to speed up operations, e.g. iteration over navigation axes. The cost of this speed improvement is to double the memory requirement during data loading.

EDAX TEAM SPD and SPC

HyperSpy can read both .spd (spectrum image) and .spc (single spectra) files from the EDAX TEAM software. If reading an .spd file, the calibration of the spectrum image is loaded from the corresponding .ipr and .spc files stored in the same directory, or from specific files indicated by the user. If these calibration files are not available, the data from the .spd file will still be loaded, but with no spatial or energy calibration. If elemental information has been defined in the spectrum image, those elements will automatically be added to the signal loaded by HyperSpy.

Currently, loading an EDAX TEAM spectrum or spectrum image will load an EDSSEMSpectrum Signal. If support for TEM EDS data is needed, please open an issue in the issues tracker to alert the developers of the need.

For further reference, file specifications for the formats are available publicly available from EDAX and are on Github (.spc, .spd, and .ipr).
Extra loading arguments for SPD file

- `spc_fname`: {None, str}, name of file from which to read the spectral calibration. If data was exported fully from EDAX TEAM software, an .spc file with the same name as the .spd should be present. If `None`, the default filename will be searched for. Otherwise, the name of the .spc file to use for calibration can be explicitly given as a string.

- `ipr_fname`: {None, str}, name of file from which to read the spatial calibration. If data was exported fully from EDAX TEAM software, an .ipr file with the same name as the .spd (plus a "_Img" suffix) should be present. If `None`, the default filename will be searched for. Otherwise, the name of the .ipr file to use for spatial calibration can be explicitly given as a string.

- `**kwargs`: remaining arguments are passed to the Numpy `memmap` function.

Extra loading arguments for SPD and SPC files

- `load_all_spc`: bool, switch to control if all of the .spc header is read, or just the important parts for import into HyperSpy.

FEI TIA SER and EMI

HyperSpy can read ser and emi files but the reading features are not complete (and probably they will be unless FEI releases the specifications of the format). That said we know that this is an important feature and if loading a particular ser or emi file fails for you, please report it as an issue in the issues tracker to make us aware of the problem.

HyperSpy (unlike TIA) can read data directly from the .ser files. However, by doing so, the information that is stored in the emi file is lost. Therefore strongly recommend to load using the .emi file instead.

When reading an .emi file if there are several .ser files associated with it, all of them will be read and returned as a list.

Extra loading arguments

- `only_valid_data`: bool, in case of series or linescan data with the acquisition stopped before the end: if True, load only the acquired data. If False, the empty data are filled with zeros. The default is False and this default value will change to True in version 2.0.

SEMPER UNF binary format

SEMPER is a fully portable system of programs for image processing, particularly suitable for applications in electron microscopy developed by Owen Saxton (see DOI: 10.1016/S0304-3991(79)80044-3 for more information). The unf format is a binary format with an extensive header for up to 3 dimensional data. HyperSpy can read and write unf-files and will try to convert the data into a fitting BaseSignal subclass, based on the information stored in the label. Currently version 7 of the format should be fully supported.

Blockfile

HyperSpy can read and write the blockfile format from NanoMegas ASTAR software. It is used to store a series of diffraction patterns from scanning precession electron diffraction (SPED) measurements, with a limited set of metadata. The header of the blockfile contains information about centering and distortions of the diffraction patterns,
but is not applied to the signal during reading. Blockfiles only support data values of type `np.uint8` (integers in range 0-255).

**Warning:** While Blockfiles are supported, it is a proprietary format, and future versions of the format might therefore not be readable. Complete interoperability with the official software can neither be guaranteed.

Blockfiles are by default loaded in a “copy-on-write” manner using `numpy.memmap`. For blockfiles `load` takes the `mmap_mode` keyword argument enabling loading the file using a different mode. However, note that lazy loading does not support in-place writing (i.e lazy loading and the “r+” mode are incompatible).

**DENS heater log**

HyperSpy can read heater log format for DENS solution’s heating holder. The format stores all the captured data for each timestamp, together with a small header in a plain-text format. The reader extracts the measured temperature along the time axis, as well as the date and calibration constants stored in the header.

**Bruker’s formats**

Bruker’s Esprit(TM) software and hardware allows to acquire and save the data in different kind of formats. Hyperspy can read two main basic formats: bcf and spx.

**Bruker composite file**

HyperSpy can read “hypermaps” saved with Bruker’s Esprit v1.x or v2.x in bcf hybrid (virtual file system/container with xml and binary data, optionally compressed) format. Most bcf import functionality is implemented. Both high-resolution 16-bit SEM images and hyperspectral EDX data can be retrieved simultaneously.

BCF can look as all inclusive format, however it does not save some key EDX parameters: any of dead/live/real times, FWHM at Mn_Ka line. However, real time for whole map is calculated from pixelAverage, lineAverage, pixelTime, lineCounter and map height parameters.

Note that Bruker Esprit uses a similar format for EBSD data, but it is not currently supported by HyperSpy.

**Extra loading arguments**

- `select_type`: one of (None, ‘spectrum’, ‘image’). If specified, only the corresponding type of data, either spectrum or image, is returned. By default (None), all data are loaded.
- `index`: one of (None, int, “all”). Allow to select the index of the dataset in the bcf file, which can contains several datasets. Default None value result in loading the first dataset. When set to ‘all’, all available datasets will be loaded and returned as separate signals.
- `downsample`: the downsampling ratio of hyperspectral array (height and width only), can be integer >=1, where ‘1’ results in no downsampling (default 1). The underlying method of downsampling is unchangeable: sum. Differently than block_reduce from skimage.measure it is memory efficient (does not creates intermediate arrays, works inplace).
- `cutoff_at_kV`: if set (can be int or float >= 0) can be used either to crop or enlarge energy (or channels) range at max values (default None).

Example of loading reduced (downsampled, and with energy range cropped) “spectrum only” data from bcf (original shape: 80keV EDS range (4096 channels), 100x75 pixels):
>>> hs.load("sample80kv.bcf", select_type='spectrum', downsample=2, cutoff_at_kV=10)
<EDSSEMSpectrum, title: EDX, dimensions: (50, 38|595)>

load the same file without extra arguments:

>>> hs.load("sample80kv.bcf")
[<Signal2D, title: BSE, dimensions: (1100, 75)>,
<Sigal2D, title: SE, dimensions: (1100, 75)>,
<EDSSEMSpectrum, title: EDX, dimensions: (100, 75|1095)>]

The loaded array energy dimension can by forced to be larger than the data recorded by setting the ‘cutoff_at_kV’ kwarg to higher value:

>>> hs.load("sample80kv.bcf", cutoff_at_kV=80)
[<Signal2D, title: BSE, dimensions: (1100, 75)>,
<Sigal2D, title: SE, dimensions: (1100, 75)>,
<EDSSEMSpectrum, title: EDX, dimensions: (100, 75|4096)>]

Note that setting downsample to >1 currently locks out using SEM imagery as navigator in the plotting.

**SPX format**

Hyperspy can read Bruker’s spx format (single spectra format based on XML). The format contains extensive list of details and parameters of EDS analyses which are mapped in hyperspy to metadata and original_metadata dictionaries.

**EMD**

EMD stands for “Electron Microscopy Dataset.” It is a subset of the open source HDF5 wrapper format. N-dimensional data arrays of any standard type can be stored in an HDF5 file, as well as tags and other metadata.

**EMD (NCEM)**

This EMD format was developed by Colin Ophus at the National Center for Electron Microscopy (NCEM). See [http://emdatasets.com/](http://emdatasets.com/) for more information.

For files containing several datasets, the **dataset_name** argument can be used to select a specific one:

```python
>>> s = hs.load("adatafile.emd", dataset_name="/experimental/science_data_1")
```

Or several by using a list:

```python
>>> s = hs.load("adatafile.emd",
... dataset_name=[
... "/experimental/science_data_1",
... "/experimental/science_data_1")
```

**EMD (Velox)**

This is a non-compliant variant of the standard EMD format developed by Thermo-Fisher (former FEI). HyperSpy supports importing images, EDS spectrum and EDS spectrum streams (spectrum images stored in a sparse format).
For spectrum streams, there are several loading options (described below) to control the frames and detectors to load and if to sum them on loading. The default is to import the sum over all frames and over all detectors in order to decrease the data size in memory.

**Note:** Pruned Velox EMD files only contain the spectrum image in a proprietary format that HyperSpy cannot read. Therefore, don’t prune FEI EMD files in you intend to read them with HyperSpy.

```python
>>> hs.load("sample.emd")
[<Signal2D, title: HAADF, dimensions: (|179, 161)>,
 <EDSSEMSpectrum, title: EDS, dimensions: (179, 161|4096)>]
```

**Note:** Currently only lazy uncompression rather than lazy loading is implemented. This means that it is not currently possible to read EDS SI Veloz EMD files with size bigger than the available memory.

**Note:** Loading a spectrum image can be slow if numba is not installed.

**Warning:** This format is still not stable and files generated with the most recent version of Velox may not be supported. If you experience issues loading a file, please report it to the HyperSpy developers so that they can add support for newer versions of the format.

### Extra loading arguments

- **select_type**: one of {None, 'image', 'single_spectrum', 'spectrum_image'} (default is None).
- **first_frame**: integer (default is 0).
- **last_frame**: integer (default is None)
- **sum_frames**: boolean (default is True)
- **sum_EDS_detectors**: boolean (default is True)
- **rebin_energy**: integer (default is 1)
- **SI_dtype**: numpy dtype (default is None)
- **load_SI_image_stack**: boolean (default is False)

The `select_type` parameter specifies the type of data to load: if `image` is selected, only images (including EDS maps) are loaded, if `single_spectrum` is selected, only single spectra are loaded and if `spectrum_image` is selected, only the spectrum image will be loaded. The `first_frame` and `last_frame` parameters can be used to select the frame range of the EDS spectrum image to load. To load each individual EDS frame, use `sum_frames=False` and the EDS spectrum image will be loaded with an an extra navigation dimension corresponding to the frame index (time axis). Use the `sum_EDS_detectors=True` parameter to load the signal of each individual EDS detector. In such a case, a corresponding number of distinct EDS signal is returned. The default is `sum_EDS_detectors=True`, which loads the EDS signal as a sum over the signals from each EDS detectors. The rebin_energy and SI_dtype parameters are particularly useful in combination with `sum_frames=False` to reduce the data size when one want to read the individual frames of the spectrum image. If SI_dtype=None (default), the dtype of the data in the emd file is used. The load_SI_image_stack parameter allows loading the stack of STEM images acquired simultaneously as the EDS spectrum image. This can be useful to monitor any specimen changes during the acquisition or to correct the spatial drift in the spectrum image by using the STEM images.
HyperSpy can read heater, biasing and gas cell log files for Protochips holder. The format stores all the captured data together with a small header in a csv file. The reader extracts the measured quantity (e.g., temperature, pressure, current, voltage) along the time axis, as well as the notes saved during the experiment. The reader returns a list of signal with each signal corresponding to a quantity. Since there is a small fluctuation in the step of the time axis, the reader assumes that the step is constant and takes its mean, which is a good approximation. Further release of HyperSpy will read the time axis more precisely by supporting non-linear axis.

1.16.4 Reading data generated by HyperSpy using other software packages

The following scripts may help reading data generated by HyperSpy using other software packages.

**ImportRPL Digital Micrograph plugin**

This Digital Micrograph plugin is designed to import Ripple files into Digital Micrograph. It is used to ease data transit between DigitalMicrograph and HyperSpy without losing the calibration using the extra keywords that HyperSpy adds to the standard format.

When executed it will ask for 2 files:

1. The riple file with the data format and calibrations
2. The data itself in raw format.

If a file with the same name and path as the riple file exits with raw or bin extension it is opened directly without prompting.

ImportRPL was written by Luiz Fernando Zagonel.

Download ImportRPL

**readHyperSpyH5 MATLAB Plugin**

This MATLAB script is designed to import HyperSpy’s saved HDF5 files (.hspy extension). Like the Digital Micrograph script above, it is used to easily transfer data from HyperSpy to MATLAB, while retaining spatial calibration information.

Download readHyperSpyH5 from its Github repository.
1.17 Events

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.

1.17.1 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:

```python
>>> 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`:

```python
>>> def on_index_changed2(index):
...     print("on_index_changed2_called")
...     print("Index: ", index)
... 
>>> s.axes_manager.indices = (3,)

on_index_changed2_called
('Index: ', 3)
... 
>>> s.axes_manager.indices = (0,)

on_index_changed2_called
('Index: ', 0)
... 
>>> def on_index_changed3():
...     print("on_index_changed3_called")
... 
>>> s.axes_manager.indices = (1,)

on_index_changed3_called
('Index: ', 1)
... 

1.17. Events
It is also possible to map trigger keyword arguments to connected function keyword arguments as follows:

```python
>>> def on_index_changed4(arg):
...     print("on_index_changed4\ncalled")
...     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)
```

### 1.17.2 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.

```python
>>> 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)
```

```python
>>> with nav_axis.events.index_changed.suppress():
...     s.axes_manager.indices = (6,)

... >>> with nav_axis.events.suppress():
...     s.axes_manager.indices = (5,)
```

### 1.17.3 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.

```python
>>> s = hs.signals.Signal1D(np.random.random((10,100)))
>>> s.data[:] = 0
>>> s.events.data_changed.trigger(obj=s)
```
1.18 Working with big data

**Warning:** All the features described in this chapter are in beta state. Although most of them work, their operation may not always be optimal, well documented and/or consistent with their in-memory counterparts. Therefore, although efforts will be taken to minimise major disruptions, the syntax and features described here may change in patch and minor HyperSpy releases. If you experience issues with HyperSpy’s lazy features please report them to the developers.

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.

1.18.1 Creating Lazy Signals

**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()`:

```python
>>> import h5py
>>> f = h5py.File("myfile.hdf5")
# Load the file
>>> data = f['/data/path']
# Get the data
>>> import dask.array as da
# Import dask to wrap
>>> chunks = (1000,100)
# Chunk as appropriate
>>> x = da.from_array(data, chunks=chunks)
# Wrap the data in dask
>>> s = hs.signals.Signal1D(x).as_lazy()
# Create the lazy signal
```

**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:

```python
>>> 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)
uint8 34.9175808
>>> s.change_dtype("float") # To be able to perform decomposition, etc.
>>> 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 \texttt{close\_file()} method. Alternatively, you could close it on calling \texttt{compute()} by passing the keyword argument \texttt{close\_file=True} e.g.:

\begin{verbatim}
>>> s = hs.load("file.hspy", lazy=True)
>>> ssum = s.sum(axis=0)
>>> ssum.compute(close_file=True)  # closes the file.hspy file
\end{verbatim}

### Lazy stacking

Occasionally the full dataset consists of many smaller files. To combine them into a one large \texttt{LazySignal}, we can \texttt{stack} them lazily (both when loading or afterwards):

\begin{verbatim}
>>> siglist = hs.load("*.hdf5")
>>> s = hs.stack(siglist, lazy=True)
>>> # Or load lazily and stack afterwards:
>>> siglist = hs.load("*.hdf5", lazy=True)
>>> s = hs.stack(siglist)  # no need to pass 'lazy', as signals already lazy
>>> # Or do everything in one go:
>>> s = hs.load("*.hdf5", lazy=True, stack=True)
\end{verbatim}

### 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 \texttt{as\_lazy()} method:

\begin{verbatim}
>>> 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)>
\end{verbatim}

### 1.18.2 Practical tips

Despite the limitations detailed below, most HyperSpy operations can be performed lazily. Important points of note are:

#### Chunking

New in version 1.3.2.

By default, HyperSpy tries to optimize the chunking for most operations. However, it is sometimes possible to manually set a more optimal chunking manually. Therefore, many operations take a \texttt{rechunk} or \texttt{optimize} keyword argument to disable automatic rechunking.

#### 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 \texttt{compute()} method:
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. A more convenient alternative is to calculate the summed navigation signal manually once, and only pass it for all other plots. Pay attention to the transpose (\( .T \)):

```python
>>> s
<LazySignal2D, title: , dimensions: (512, 512)>
>>> s.compute()
[########################################] | 100% Completed | 0.1s
>>> s
<Signal2D, title: , dimensions: (512, 512)>
```

# for fastest results, just pick one signal space pixel

```python
>>> nav = s.transpose(optimize=True).inav[256, 256]
>>> nav = s.sum(s.axes_manager.signal_axes).T
```

Alternatively, it is possible to not have a navigator, and use sliders instead:

```python
>>> s
<LazySignal2D, title: , dimensions: (200, 200|512, 512)>
>>> s.plot(navigator='slider')
```

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 \( \text{shift1D()} \) is not lazy.

1.18.3 Limitations

Most operations can be performed lazily. However, lazy operations come with a few limitations and constraints that we detail below.

Immutable signals

An important limitation when using \texttt{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:

```python
>>> 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:

```python
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s = s + 1
```

Or even better:

```python
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s1 = s + 1
```

### Machine learning (decomposition)

*Decomposition* algorithms often performs large matrix manipulations, requiring significantly more memory than the data size. To perform decomposition operation lazily HyperSpy provides several “online” algorithms and *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:

- **PCA** (`algorithm='PCA'`): performs IncrementalPCA from *scikit-learn*. Please refer to its documentation for a description of the several keyword arguments taken by its `fit` method.
- **ORPCA** (`algorithm='ORPCA'`): performs Online Robust PCA. Please refer to the docstring of `ORPCA()` for details on usage and keyword arguments.
- **NMF** (`algorithm='ONMF'`): performs Online Robust NMF, as per “OPGD” algorithm in [Zhao2016]. Please refer to the docstring of `ONMF()` for details on usage and keyword arguments.

### 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.

### 1.18.4 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` (more information here). `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` (documentation here) 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.

### 1.19 Metadata structure

The `BaseSignal` class stores metadata in the `metadata` attribute that has a tree structure. By convention, the nodes 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.

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.
1.19.1 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.

1.19.2 Acquisition_instrument

**TEM**

Contain information relevant to transmission electron microscope signals.

**microscope** type: Str

The microscope model, e.g. VG 501

**acquisition_mode** type: Str

Either ‘TEM’ or ‘STEM’

**camera_length** type: Float

The camera length in mm.
**convergence_angle**  type: Float
The beam convergence semi-angle in mrad.

**beam_energy**  type: Float
The energy of the electron beam in keV

**beam_current**  type: Float
The beam current in nA.

**probe_area**  type: Float
The illumination area of the electron beam in nm$^2$.

**dwell_time**  type: Float
The dwell time in seconds. This is relevant for STEM acquisition

**exposure**  type: Float
The exposure time in seconds. This is relevant for TEM acquisition.

**magnification**  type: Float
The magnification.

**SEM**

Contain information relevant to scanning electron microscope signals.

**microscope**  type: Str
The microscope model, e.g. VG 501

**convergence_angle**  type: Float
The beam convergence semi-angle in mrad.

**beam_energy**  type: Float
The energy of the electron beam in keV

**beam_current**  type: Float
The beam current in nA.

**probe_area**  type: Float
The illumination area of the electron beam in nm$^2$.

**magnification**  type: Float
The magnification.

**working_distance**  type: Float
The working distance in mm.

**Stage**

**tilt_alpha**  type: Float
A tilt of the stage in degree.
**tilt_beta**  type: Float

Another tilt of the stage in degree.

**rotation**  type: Float

The rotation of the stage in degree.

**x**  type: Float

The position of the stage in mm along the x axis.

**y**  type: Float

The position of the stage in mm along the y axis.

**z**  type: Float

The position of the stage in mm along the z axis.

**Detector**

All instruments can contain a “Detector” node with information about the detector used to acquire the signal. EDX and EELS detectors should follow the following structure:

**detector_type**  type: Str

The type of the detector, e.g. SE for SEM

**EELS**

This node stores parameters relevant to electron energy loss spectroscopy signals.

**aperture_size**  type: Float

The entrance aperture size of the spectrometer in mm.

**collection_angle**  type: Float

The collection semi-angle in mrad.

**dwell_time**  type: Float

The dwell time in seconds. This is relevant for STEM acquisition

**exposure**  type: Float

The exposure time in seconds. This is relevant for TEM acquisition.

**frame_number**  type: int

The number of frames/spectra integrated during the acquisition.

**spectrometer**  type: Str

The spectrometer model, e.g. Gatan Enfinium ER (Model 977).

**EDS**

This node stores parameters relevant to electron X-ray energy dispersive spectroscopy data.

**azimuth_angle**  type: Float

The azimuth angle of the detector in degree. If the azimuth is zero, the detector is perpendicular to the tilt axis.
The elevation angle of the detector in degree. The detector is perpendicular to the surface with an angle of 90.

The full width at half maximum (FWHM) of the manganese K alpha (Mn Ka) peak in eV. This value is used as a first approximation of the energy resolution of the detector.

The time spent to record the spectrum in second.

The time spent to record the spectrum in second, compensated for the dead time of the detector.

This node stores parameters of biprism used in off-axis electron holography

Rotation angle of the biprism in degree

Position of the biprism in microscope column, e.g. Selected area aperture plane

Voltage of electrostatic biprism in volts

Acknowledgment of sample supplier, e.g. Prepared by Putin, Vladimir V.

A brief description of the sample

A list of the symbols of the elements composing the sample, e.g. ['B', 'N'] for a sample composed of Boron and Nitrogen.

A list of the symbols of the X-ray lines to be used for processing, e.g. ['Al_Ka', 'Ni_Lb'] for the K alpha line of Aluminum and the L beta line of Nickel.

The thickness of the sample in m.

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: \( \text{variance} = (\text{gain\_factor} \times \text{data} + \text{gain\_offset}) \times \text{correlation\_factor} \).

**gain\_factor** type: Float

**gain\_offset** type: Float

**correlation\_factor** type: Float

**parameters\_estimation\_method** type: Str

**1.19.5 Internal parameters**

This node is “private” and therefore is not displayed when printing the `metadata` attribute. For example, an “energy” leaf should be accompanied by an “energy\_units” leaf.

**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

Contains parameters that related to the folding/unfolding of signals.

1.20 Bibliography

1.20.1 Bibliography


1.20.2 Peer-review articles with results obtained using HyperSpy

Note: Given the increasing number of articles that cite HyperSpy we no longer maintain a list of articles here. For an up-to-date list search for HyperSpy in a scientific database e.g. Google Scholar.

Warning: The articles published before 2012 may mention the HyperSpy project under its old name, EELSLab.
2.1 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.

2.1.1 Getting started

2.1.2 1. Start using HyperSpy and understand it

Probably you would not be interested in contributing to HyperSpy if you were not already an 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 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: HyperSpy User-Guide.

2.1.3 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.

2.1.4 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’ll 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.

### 2.1.5 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 part 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. Beside, 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’s 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!

### 2.2 Running and writing tests

#### 2.2.1 Write tests

Every new function that is written in to HyperSpy needs to be tested and documented.

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 than 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’re very important as they force you to sanity check all you do. For details on running and writing HyperSpy test see dev_tests

HyperSpy uses the pytest library for testing. The tests reside in the hyperspy.tests module.

First ensure pytest and its plugins are installed by:

```bash
# If using a standard hyperspy install
pip install hyperspy[test]
# Or, from a hyperspy local development directory
```

(continues on next page)
To run them:

```
pip install -e .[test]
# Or just installing the dependencies using conda
conda install -c conda-forge pytest pytest-mpl
```

Or, from HyperSpy’s project folder simply:

```
pytest
```

Useful hints on testing:

- When comparing integers, it's fine to use ``==``. When comparing floats, be sure to use numpy.testing.
  
  ```
  assert_almost_equal() or numpy.testing.assert_allclose().
  ```

- `numpy.testing.assert_equal()` is convenient to compare 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 does not use too much of 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 to the `s` object, for example:

  ```
  del s
  gc.collect()
  ```

- Once, you have pushed your PR to the official HyperSpy repository, it can be useful to check the coverage of your tests using the coveralls.io check of your PR. This service can help you to find how well your code is being tested and exactly which part is not currently tested.

- `pytest-sugar` can be installed to have a nicer look and feel of pytest in the console (encoding issue have been reported in the Windows console).

### 2.2.2 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 references images are located in the folder defined by the argument `baseline_dir` of the `@pytest.mark.mpl_image_compare` decorator.

To run plotting tests, you simply need to add the option `--mpl`:

```
pytest --mpl
```

If you don’t use the `--mpl`, the code of the tests will be executed but the images will not be compared to the references images.

If you need to add or change some plots, follow the workflow below:

1. Write the tests using appropriate decorator such as `@pytest.mark.mpl_image_compare`.  

2. If you need to generate 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 are failing, it is possible to download the figure comparison images generated by pytest-mpl in the `artifacts` tabs of the corresponding build.

**The plotting tests need matplotlib > 3.0.0, since small changes in the way** matplotlib generates the figure can make the tests fail.

In travis and appveyor, 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 set accordingly the backend.

See `pytest-mpl` for more details.

### 2.2.3 Exporting pytest results as HTML

With `pytest-html` it is possible to export the results of running pytest for easier viewing. I can be installed by conda:

```bash
conda install pytest-html
```

and run by:

```bash
pytest --mpl --html=report.html
```

### 2.3 Writing documentation

Documentation comes in two parts: docstrings and user-guide documentation.

Docstrings – written at the start of a function and 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.

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 kind of data which the users can use themselves. Artificial datasets for this purpose can be found in `hyperspy.datasets.artificial_data`.

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:

```bash
$ conda install hyperspy-dev
```

or

```bash
$ pip install hyperspy[build-doc]
```

When writing documentation the Python package `sphinx.inv` can be useful for writing cross-references. For example, to find how to write a cross-reference to `:py:method:`hyperspy.signal.BaseSignal.set_signal_type`:
$ sphobjinv suggest doc/_build/html/objects.inv set_signal_type -st 90

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
</table>

## 2.4 Coding style

HyperSpy follows the Style Guide for Python Code - these are just some rules for consistency that you can read all about in the Python Style Guide.

You can check your code with the pep8 Code Checker.

Additionally you could use autopep8 to fix the style of your code automatically. In Linux and MacOS you can run autopep8 automatically after each commit by adding a post-commit file to .git/hooks with the following content:

```bash
#!/bin/sh
# From https://gist.github.com/temoto/6183235
FILES=$(git diff HEAD^ HEAD --name-only --diff-filter=ACM | grep -e '\.py$')
if [ -n "$FILES" ]; then
  for f in $FILES
  do
    # auto pep8 correction
    autopep8 --in-place -v --aggressive $f
    git add $f
  done
  #git commit -m "Automatic style corrections courtesy of autopep8"
  GIT_COMMITTER_NAME="autopep8" GIT_COMMITTER_EMAIL="autopep8@email.com" git commit --author="autopep8 <autopep8@email.com>" -m "Automatic style corrections courtesy of autopep8"
```

## 2.5 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 – 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.

The easiest way to add new methods that work both with arbitrary navigation dimensions and LazySignals is by using the `map` (or, for more control, `_map_all` or `_map_iterate`) 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.
If the new method cannot be coerced into a shape suitable for \texttt{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, \texttt{da.map_blocks} and \texttt{da.map_overlap} are efficient and flexible.

Finally, in addition to \texttt{iterate_signal} that is available to all HyperSpy signals, lazy counterparts also have \texttt{block_iter} 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 \texttt{dask} documentation.

2.6 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.

2.6.1 Writing Numba code

If you need to improve the speed of a given part of the code your first choice should be Numba. This is because Numba code is very similar (when not identical) to Python code, and, therefore, it is a lot easier to maintain than Cython code.

2.6.2 Writing Cython code

Cython code should only be considered if:

1. It is not possible to speed up the function using numba 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 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 they are typically very large.

Once you have written your Cython files, add them to \texttt{raw_extensions} in \texttt{setup.py}.

Compiling Cython code

If Cython is present in the build environment and any cythonized c/c++ file is missing, then \texttt{setup.py} tries to cythonize all extensions automatically.

To make the development easier \texttt{setup.py} provides a \texttt{recythonize} command that can be used in conjunction with default commands. For example \texttt{python setup.py recythonize build_ext --inplace} will re-cythonize all Cython code and compile it.

Cythonization and compilation also takes place during continuous integration (CI).
2.7 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 be provided by external packages too but don’t need to be registered. Instead, they are returned by the `create_model` method of the relevant `hyperspy.signal.BaseSignal` subclass, see for example, the `hyperspy._signals.eds_tem.EDSTEM_mixin.create_model()` of the `EDSTEMSpectrum`.

It is good practice to add all packages that extend HyperSpy to the list of known extensions regardless their maturity level. In this way we can avoid duplication of efforts and issues arising from naming conflicts.

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 software ecosystem.

### 2.7.1 Registering extensions

In order to register HyperSpy extensions you need to:

1. Add the following line to your package’s `setup.py`:

   ```python
   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.

### 2.7.2 Creating new HyperSpy BaseSignal subclasses

**When and where create a new BaseSignal subclass**

HyperSpy provides most of its functionality through the different `hyperspy.signal.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 `hyperspy.signal.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.
New function needed
Is it useful for data of any type and dimensions?
Contribute it to BaseSignal
Yes
Does an SignalxD for the required dimension exist in HyperSpy
No
Contribute new SignalxD to HyperSpy
No
Is the function useful only for some sort of data?
Yes
Contribute it to SignalxD
No
Does an signal for that sort of data exists?
Yes
Contribute to package providing the relevant signal
Yes
Create you own package and signal subclass to host the function
No

Registering a new BaseSignal subclass

To register a new `hyperspy.signal.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 packages 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 `hyperspy.signal.BaseSignal.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.`
2.7.3 Creating new HyperSpy model components

When and where 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`. For your function you should consider creating your own.

```
New component needed
  Can it be declared using Expression?
    Yes
    Can it be useful to other users?
      Yes
      Create new component from the scratch
      No
      Create new component using Expression
    No
    Just use Expression
  No
  Is it useful for general users?
    Yes
    Contribute it to HyperSpy
    No
    Does a suitable package for it exist?
      Yes
      Contribute it to the relevant package
      No
      Create your own package to host it
```

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 an 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:

2.7. Writing packages that extend HyperSpy
components2D:
# _id_name of the component. It must be an 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 an 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 an 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.

2.7.4 Creating and registering new widgets and toolkeys

To generate GUIs of specific method and functions, HyperSpy use widgets and toolkeys:

- widgets (typically ipywidgets or traitsui objects) generate GUIs,
- toolkeys are functions to which it is possible to 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:

```yaml
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 the extensions. In order to avoid name clashes, by convention
  # toolkeys must start by the name of the packages 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.

```yaml
GUI:
  widgets:
    ipywidgets:
```
(continues on next page)
Each widget is declared using a dictionary with two keys, `module` and `function`.

```python
my_package.MyComponent:
    # The function that creates the widget
    function: get_mycomponent_widget
    # The module where the function resides.
    module: my_package.widgets
```
3.1 hyperspy package

3.1.1 Subpackages

hyperspy._components package

Submodules

hyperspy._components.arctan module

class hyperspy._components.arctan.Arctan(A=1.0, k=1.0, x0=1.0, minimum_at_zero=False)
   Bases: hyperspy.component.Component

   Arctan function component.

   \[ f(x) = A \cdot \arctan [k (x - x_0)] \]

   Parameters

   - \( A \) (float)
   - \( k \) (float)
   - \( x_0 \) (float)

   function(x)
   grad_A(x)
   grad_k(x)
   grad_x0(x)
class hyperspy._components.bleasdale.Bleasdale(a=1.0, b=1.0, c=1.0, module='numexpr', **kwargs)

Bases: hyperspy._components.expression.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) –
- **b** (Float) –
- **c** (Float) –
- **kwargs** – Extra keyword arguments are passed to the Expression component.

For \((a + b \cdot x) \leq 0\), the component will be set to 0.

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._components.eels_cl_edge.EELSCLEdge(element_subshell, GOS=None)

Bases: hyperspy.component.Component

EELS core loss ionisation edge from hydrogenic or tabulated Hartree-Slater GOS with splines for fine structure fitting.

Hydrogenic GOS are limited to K and L shells.

Currently it only supports Peter Rez’s Hartree Slater cross sections parametrised as distributed by Gatan in their Digital Micrograph (DM) software. If Digital Micrograph is installed in the system HyperSpy in the standard location HyperSpy should find the path to the HS GOS folder. Otherwise, the location of the folder can be defined in HyperSpy preferences, which can be done through hs.preferences.gui() or the hs.preferences.EELS.eels_gos_files_path variable.

Parameters

- **element_subshell**(str, dict) – Usually a string, for example, ‘Ti_L3’ for the GOS of the titanium L3 subshell. If a dictionary is passed, it is assumed that Hartree Slater GOS was exported using GOS.as_dictionary, and will be reconstructed.

- **GOS**(hydrogenic, 'Hartree-Slater', None) – The GOS to use. If None it will use the Hartree-Slater GOS if they are available, otherwise it will use the hydrogenic GOS.

onset_energy

The edge onset position
Type Parameter

**intensity**
The factor by which the cross section is multiplied, what in favourable cases is proportional to the number of atoms of the element. It is a component.Parameter instance. It is fixed by default.

Type Parameter

**fine_structure_coeff**
The coefficients of the spline that fits the fine structure. Fix this parameter to fix the fine structure. It is a component.Parameter instance.

Type Parameter

**effective_angle**
The effective collection semi-angle. It is automatically calculated by set_microscope_parameters. It is a component.Parameter instance. It is fixed by default.

Type Parameter

**fine_structure_smoothing**
Controls the level of smoothing of the fine structure model. Decreasing the value increases the level of smoothing.

Type float between 0 and 1

**fine_structure_active**
Activates/deactivates the fine structure feature.

Type bool

**property E0**
**property collection_angle**
**property convergence_angle**
**property fine_structure_active**

**fine_structure_coeff_to_txt** *(filename)*

**property fine_structure_smoothing**
Controls the level of the smoothing of the fine structure.

It must a real number between 0 and 1. The higher close to 0 the higher the smoothing.

**property fine_structure_width**

**function** *(E)*
Returns the number of counts in barns

**get_fine_structure_as_signal1D()**
Returns a spectrum containing the fine structure.

**Notes**
The fine structure is corrected from multiple scattering if the model was convolved with a low-loss spectrum

**grad_intensity** *(E)*

**gui** *(display=True, toolkit=None, **kwargs)*
Display or return interactive GUI element if available.

**Parameters**

3.1. hyperspy package
• **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 strings 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.

**set_microscope_parameters** *(E0, alpha, beta, energy_scale)*

Parameters

• **E0** *(float)* – Electron beam energy in keV.

• **alpha** *(float)* – Convergence semi-angle in mrad.

• **beta** *(float)* – Collection semi-angle in mrad.

• **energy_scale** *(float)* – The energy step in eV.

**hyperspy_components.eels_double_power_law module**

**class hyperspy_components.eels_double_power_law.DoublePowerLaw** *(A=1e-05, r=3.0, origin=0.0, shift=20.0, ratio=1.0, left_cutoff=0.0, module='numexpr', compute_gradients=False, **kwargs)*

Bases: **hyperspy_components.expression.Expression**

Double power law component for EELS spectra.

\[ f(x) = A \cdot (s_r \cdot (x - x_0 - x_s)^{-r} + (x - x_0)^{-r}) \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>x_0</td>
<td>origin</td>
</tr>
<tr>
<td>x_s</td>
<td>shift</td>
</tr>
<tr>
<td>s_r</td>
<td>ratio</td>
</tr>
</tbody>
</table>

Parameters

• **A** *(float)* – Height parameter.

• **r** *(float)* – Power law coefficient.

• **origin** *(float)* – Location parameter.

• **shift** *(float)* – Offset of second power law.

• **ratio** *(float)* – Height ratio of the two power law components.
• **kwargs – Extra keyword arguments are passed to the Expression component.

The left_cutoff parameter can be used to set a lower threshold from which the component will return 0.

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 obtain the fitted component without iterating over the navigation axes.

grad_A(x)

grad_origin(x)

grad_r(x)

grad_ratio(x)

grad_shift(x)

hyperspy._components.eels_vignetting module

class hyperspy._components.eels_vignetting.Vignetting
    Bases: hyperspy.component.Component

Model the vignetting of the lens with a cos^4 law multiplied by lines on the edges

fix_cos_vignetting()

fix_side_vignetting()

free_cos_vignetting()

free_side_vignetting()

function(x)

hyperspy._components.error_function module

class hyperspy._components.error_function.Erf(A=1.0, sigma=1.0, origin=0.0, module='scipy', **kwargs)
    Bases: hyperspy._components.expression.Expression

Error function component.

\[ f(x) = \frac{A}{2} \text{erf} \left( \frac{x - x_0}{\sqrt{2} \sigma} \right) \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>\sigma</td>
<td>sigma</td>
</tr>
<tr>
<td>x_0</td>
<td>origin</td>
</tr>
</tbody>
</table>

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.
class hyperspy._components.exponential.Exponential(A=1.0, tau=1.0, module='numexpr', **kwargs)

Bases: hyperspy._components.expression.Expression

Exponential function component.

\[ f(x) = A \cdot \exp \left( -\frac{x}{\tau} \right) \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>( \tau )</td>
<td>( \tau )</td>
</tr>
</tbody>
</table>

Parameters

- **A** (float) – Maximum intensity
- **tau** (float) – Scale parameter (time constant)
- ****kwargs – Extra keyword arguments are passed to the Expression component.

class hyperspy._components.expression.Expression(expression, name, position=None, module='numpy', autodoc=True, add_rotation=False, rotation_center=None, rename_pars={}, compute_gradients=True, **kwargs)

Bases: hyperspy.component.Component

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

- **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** [[“numpy”, “numexpr”], default “numpy”] Module used to evaluate the function. numexpr is often faster but it supports fewer functions and requires installing numexpr.

- **add_rotation**: bool, default False This is only relevant for 2D components. If True it automatically adds rotation_angle parameter.

- **rotation_center** [[None, 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: dictionary  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.

**kwargs  Keyword arguments can be used to initialise the value of the parameters.

recompile: useful to recompile the function and gradient with a different module.

Note:  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:

```python
>>> 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",)
```

Substitutions for long or complicated expressions are separated by semicolons:

```python
>>> 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>)
```

**kwargs  Keyword arguments can be used to initialise the value of the parameters.

':

compile_function (module='numpy', position=False)

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.

hyperspy_components gaussian module

class hyperspy_components gaussian Gaussian(A=1.0, sigma=1.0, centre=0.0, module='numexpr', **kwargs)

Bases: hyperspy_components expression.Expression
Normalized Gaussian function component.

\[ f(x) = \frac{A}{\sigma \sqrt{2\pi}} \exp \left[ -\frac{(x-x_0)^2}{2\sigma^2} \right] \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>sigma</td>
</tr>
<tr>
<td>(x_0)</td>
<td>centre</td>
</tr>
</tbody>
</table>

Parameters

- **A** (**float**) – 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.

For convenience the **fwhm** attribute can be used to get and set the full-with-half-maximum.

See also:

`hyperspy._components.gaussianhf.GaussianHF`

**estimate_parameters** (*signal, x1, x2, only_current=False*)

Estimate the Gaussian by calculating the momenta.

Parameters

- **signal** (**Signal1D instance**) –
- **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

Return type **bool**

Notes

Adapted from [http://www.scipy.org/Cookbook/FittingData](http://www.scipy.org/Cookbook/FittingData)

Examples

```python
>>> 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._axes[-1].offset = -10
>>> s.axes_manager._axes[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
```

property **fwhm**
**hyperspy_components.gaussian2d module**

```python
class hyperspy_components.gaussian2d.Gaussian2D(A=1.0, sigma_x=1.0, sigma_y=1.0, centre_x=0.0, centre_y=0, module='numexpr', **kwargs)

Bases: hyperspy_components.expression.Expression

Normalized 2D elliptical Gaussian function component.

\[
f(x, y) = \frac{A}{2\pi s_x s_y} \exp \left[ -\frac{(x - x_0)^2}{2s_x^2} - \frac{(y - y_0)^2}{2s_y^2} \right]
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>s_x, s_y</td>
<td>sigma_x/y</td>
</tr>
<tr>
<td>x_0, y_0</td>
<td>centre_x/y</td>
</tr>
</tbody>
</table>

**Parameters**

- **A (float)** – Amplitude (height of the peak scaled by \(2\pi s_x s_y\)).
- **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.

For convenience the \(fwhm_x\) and \(fwhm_y\) attributes can be used to get and set the full-with-half-maxima along the two axes.

```python
property fwhm_x
property fwhm_y
```

**hyperspy_components.gaussianhf module**

```python
class hyperspy_components.gaussianhf.GaussianHF(height=1.0, fwhm=1.0, centre=0.0, module='numexpr', **kwargs)

Bases: hyperspy_components.expression.Expression

Normalized gaussian function component, with a \(fwhm\) parameter instead of the \(sigma\) parameter, and a \(height\) parameter instead of the \(A\) parameter (scaling difference of \(\sigma \sqrt{(2\pi)}\)). This makes the parameter vs. peak maximum independent of \(\sigma\), 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 (\(signal_axis.scale\)).

\[
f(x) = h \cdot \exp \left[ -\frac{4 \log 2 (x - c)^2}{W^2} \right]
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>height</td>
</tr>
<tr>
<td>W</td>
<td>fwhm</td>
</tr>
<tr>
<td>c</td>
<td>centre</td>
</tr>
</tbody>
</table>
```
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.

The helper properties *sigma* and *A* are also defined for compatibility with *Gaussian* component.

See also:

*hyperspy._components.gaussian.Gaussian*

**property A**

**estimate_parameters** *(signal, x1, x2, only_current=False)*

Estimate the gaussian by calculating the momenta.

Parameters

- **signal** *(Signal1D instance)* –
- **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

Return type  bool

Notes

Adapted from http://www.scipy.org/Cookbook/FittingData

Examples

```python
>>> 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)

integral_as_signal()

Utility function to get gaussian integral as Signal1D

**property sigma**
**hyperspy\_components.heaviside module**

```python
class hyperspy\_components.heaviside.HeavisideStep(A=1, n=0)
    Bases: hyperspy.component.Component

    The Heaviside step function.

    \[ f(x) = \begin{cases} 
    0 & x < n \\
    \frac{A}{2} & n \leq x \
    A & x > n 
    \end{cases} \]

    Parameters
    - \( n \) (float) -
    - \( A \) (float) -
```

```python
function(x)
grad_A(x)
grad_n(x)
```

**hyperspy\_components.logistic module**

```python
class hyperspy\_components.logistic.Logistic(a=1.0, b=1.0, c=1.0, origin=0.0, module='numexpr', **kwargs)
    Bases: hyperspy\_components.expression.Expression

    Logistic function (sigmoid or s-shaped curve) component.

    \[ f(x) = \frac{a}{1 + b \cdot \exp\left[-c((x-x_0)]\right) \]

    Parameters
    - \( a \) (Float) – The curve’s maximum y-value, \( \lim_{x \to \infty} (y) = a \)
    - \( b \) (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
    - \( \text{origin} \) (Float) – Position of the sigmoid’s midpoint
    - **kwargs – Extra keyword arguments are passed to the Expression component.
```

### 3.1. hyperspy package

203
**hyperspy._components.lorentzian module**

```python
class hyperspy._components.lorentzian.Lorentzian(A=1.0, gamma=1.0, centre=0.0, module='numexpr', **kwargs)
```

Cauchy-Lorentz distribution (a.k.a. Lorentzian function) component.

\[
f(x) = \frac{A}{\pi} \frac{\gamma}{(x - x_0)^2 + \gamma^2}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>gamma</td>
</tr>
<tr>
<td>(x_0)</td>
<td>centre</td>
</tr>
</tbody>
</table>

**Parameters**

- \(A\) (float) – Height parameter, where \(A/(\gamma \pi)\) is the maximum of the peak.
- \(\text{gamma}\) (float) – Scale parameter corresponding to the half-width-at-half-maximum of the peak, which corresponds to the interquartile spread.
- \(\text{centre}\) (float) – Location of the peak maximum.
- \(**kwargs\) – Extra keyword arguments are passed to the Expression component.

For convenience the \(fwhm\) attribute can be used to get and set the full-with-half-maximum.

```python
property fwhm
```

**hyperspy._components.offset module**

```python
class hyperspy._components.offset.Offset(offset=0.0)
```

Component to add a constant value in the y-axis.

\[
f(x) = k + x
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k)</td>
<td>offset</td>
</tr>
</tbody>
</table>

**Parameters**

- \(\text{offset}\) (float) –

**estimate_parameters** (signal, \(x1, x2, only\_current=False\))

Estimate the parameters by the two area method

**Parameters**

- \(\text{signal}\) (BaseSignal instance) –
- \(\text{x1}\) (float) – Defines the left limit of the spectral range to use for the estimation.
- \(\text{x2}\) (float) – Defines the right limit of the spectral range to use for the estimation.
- \(\text{only\_current}\) (bool) – If False estimates the parameters for the full dataset.
Returns

Return type  bool

function(x)

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.

static grad_offset(x)

hyperspy_components.pes_core_line_shape module

class hyperspy_components.pes_core_line_shape.PESCoreLineShape(A=1.0, FWHM=1.0, origin=0.0)

Bases: hyperspy.component.Component

function(x)
   Given an one dimensional array x containing the energies at which you want to evaluate the background model, returns the background model for the current parameters.

grad_A(x)
grad_FWHM(x)
grad_ab(x)
grad_origin(x)

hyperspy_components.pes_see module

class hyperspy_components.pes_see.SEE(A=1.0, Phi=1.0, B=0.0, sigma=0)

Bases: hyperspy.component.Component

Secondary electron emission component for Photoemission Spectroscopy

A
   Type  float

Phi
   Type  float

B
   Type  float

sigma
   Resolution parameter.
      Type  float

function(x)
grad_A(x)
grad_B(x)
grad_Phi(x)
grad_sigma(x)
HyperSpy Documentation, Release 1.5.1.dev

**hyperspy._components.polynomial module**

```python
class hyperspy._components.polynomial.Polynomial(order=2, module='numexpr', **kwargs)
```

Bases: `hyperspy._components.expression.Expression`

-n-order polynomial component.

Polynomial component consisting of order + 1 parameters. The parameters are named “a” followed by the corresponding order, i.e.

\[ f(x) = a_2 x^2 + a_1 x + a_0 \]

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.

**estimate_parameters** *(signal, x1, x2, only_current=False)*

Estimate the parameters by the two area method

**Parameters**

- **signal** *(Signal1D instance)* –
- **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**

- **Return type** bool

**get_polynomial_order** ()

**hyperspy._components.polynomial.convert_to_polynomial** *(poly_dict)*

Convert the dictionary from the old to the new polynomial definition

**hyperspy._components.polynomial_deprecated module**

```python
class hyperspy._components.polynomial_deprecated.Polynomial(order=2, legacy=True, module='numexpr', **kwargs)
```

Bases: `hyperspy.component.Component`

-n-order polynomial component. (DEPRECATED) Polynomial component defined by the coefficients parameters which is an array of len the order of the polynomial. For example, the [1,2,3] coefficients define the following 3rd order polynomial: \( f(x) = 1x^2 + 2x + 3 \) Polynomial will be replaced by Polynomial2

This API is deprecated and will be replaced by `hyperspy._components.polynomial.Polynomial` in HyperSpy v2.0. To use the new API, set legacy to False.

**coefficients**

- **Type** array
Polynomial component (DEPRECATED)

This API is deprecated and will be replaced by `hyperspy._components.polynomial.Polynomial` in HyperSpy v2.0. To use the new API, set `legacy` to `False`.

**Parameters**
- `order (int)` – Order of the polynomial.
- `legacy (bool, default True)` – If `False`, use the new API.
- `module (str)` – See the docstring of `hyperspy._components.polynomial.Polynomial` for details.

**estimate_parameters** `(signal, x1, x2, only_current=False)`
Estimate the parameters by the two area method.

**Parameters**
- `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**
- `function (x)`
- `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 obtain the fitted component without iterating over the navigation axes.

- `get_polynomial_order ()`
- `grad_coefficients (x)`
- `grad_one_coefficient (x, index)`

Returns the gradient of one coefficient

**hyperspy._components.power_law module**

**class hyperspy._components.power_law.PowerLaw**

```
class hyperspy._components.power_law.PowerLaw(A=1000000.0, r=3.0, origin=0.0, left_cutoff=0.0, module='numexpr', compute_gradients=False, **kwargs)
```

**Bases:** `hyperspy._components.expression.Expression`

Power law component.

\[ f(x) = A \cdot (x - x_0)^{-r} \]

**Parameters**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>x₀</td>
<td>origin</td>
</tr>
</tbody>
</table>
• $A$ \((float)\) – Height parameter.
• $r$ \((float)\) – Power law coefficient.
• $\text{origin}$ \((float)\) – Location parameter.
• $**\text{kwargs}$ – Extra keyword arguments are passed to the Expression component.

The left\_cutoff parameter can be used to set a lower threshold from which the component will return 0.

$\text{estimate\_parameters}$ \((signal, x1, x2, only\_current=False, out=False)\)

Estimate the parameters for the power law component by the two area method.

Parameters

• $signal$ \((Signal1D\ \text{instance})\) –
• $x1$ \((float)\) – 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

Return type  \{bool, tuple of values\}

$\text{grad\_A}(x)$
$\text{grad\_origin}(x)$
$\text{grad\_r}(x)$

$\text{hyperspy\_components\_rc module}$

$\text{class hyperspy\_components\_rc.RC(}Vmax=1.0, V0=0.0, tau=1.0, module='numexpr', **kwargs\text{)}$

Bases: $\text{hyperspy\_components\_expression.Expression}$

RC function component (based on the time-domain capacitor voltage response of an RC-circuit)

$$f(x) = V_0 + V_{\text{max}} \left[ 1 - \exp \left( -\frac{x}{\tau} \right) \right]$$

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{\text{max}}$</td>
<td>$V_{\text{max}}$</td>
</tr>
<tr>
<td>$V_0$</td>
<td>$V_0$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>$\tau$</td>
</tr>
</tbody>
</table>

Parameters

• $V_{\text{max}}$ \((float)\) – maximum voltage, asymptote of the function for $\lim_{x \to \infty}$
• $V_0$ \((float)\) – vertical offset
• $\tau$ \((float)\) – $\tau=RC$ is the RC circuit time constant (voltage rise time)
• $**\text{kwargs}$ – Extra keyword arguments are passed to the Expression component.
The fixed pattern is defined by a single spectrum which must be provided to the ScalableFixedPattern constructor, e.g.:

```python
In [1]: s = load('my_spectrum.hspy')
In [2]: my_fixed_pattern = components.ScalableFixedPattern(s))
```

### Parameters

- **yscale** *(Float)* –
- **xscale** *(Float)* –
- **shift** *(Float)* –
- **interpolate** *(Bool)* – If False no interpolation is performed and only a y-scaled spectrum is returned.

### prepare_interpolator : method to fine tune the interpolation function *(x)*

### grad_yscale *(x)*

### 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 strings 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(kind='linear', fill_value=0, **kwargs)

Prepare interpolation.

**Parameters**

- **x (array)** – The spectral axis of the fixed pattern
- **kind (str or int, optional)** – 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. Default is ‘linear’.
- **fill_value (float, optional)** – If provided, then this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN.

**Notes**

Any extra keyword argument is passed to scipy.interpolate.interp1d

**hyperspy._components.skew_normal module**

class hyperspy._components.skew_normal.SkewNormal(x0=0.0, A=1.0, scale=1.0, shape=0.0, module='scipy', **kwargs)

Bases: hyperspy._components.expression.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/

\[
\begin{align*}
    f(x) &= 2A\phi(x)\Phi(x) \\
    \phi(x) &= \frac{1}{\sqrt{2\pi}}\exp\left[-\frac{t(x)^2}{2}\right] \\
    \Phi(x) &= \frac{1}{2}\left[1 + \text{erf}\left(\frac{\alpha t(x)}{\sqrt{2}}\right)\right] \\
    t(x) &= \frac{x - x_0}{\omega}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)</td>
<td>x0</td>
</tr>
<tr>
<td>(A)</td>
<td>A</td>
</tr>
<tr>
<td>(\omega)</td>
<td>scale</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>shape</td>
</tr>
</tbody>
</table>

**Parameters**

- **\(x_0\) (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.

The properties *mean* (position), *variance*, *skewness* and *mode* (=position of maximum) are defined for convenience.

```python
property mean
property mode
property skewness
property variance
```

**hyperspy._components.voigt module**

```python
class hyperspy._components.voigt.Voigt
```

Voigt profile component with support for shirley background, non_isochromaticity, transmission_function corrections and spin orbit splitting specially suited for Photoemission spectroscopy data analysis.

\[ f(x) = G(x) \times L(x) \]

where \( G(x) \) is the Gaussian function and \( L(x) \) is the Lorentzian function.

```python
area
centre
FWHM
gamma
resolution
shirley_background
non_isochromaticity
transmission_function
spin_orbit_splitting
spin_orbit_branching_ratio
```

```python
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type Parameter
Type float
```
spin_orbit_splitting_energy

Type float

estimate_parameters (signal, E1, E2, only_current=False)

Estimate the voigt function by calculating the momenta the gaussian.

Parameters

• signal (Signal1D instance) –
• 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

Return type bool

Notes

Adapted from http://www.scipy.org/Cookbook/FittingData

Examples

```python
>>> 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' : data})
>>> s.axes_manager.axes[-1].offset = -10
>>> s.axes_manager.axes[-1].scale = 0.01
>>> g.estimate_parameters(s, -10,10, False)
```

Voigt lineshape.

The voigt peak is the convolution of a Lorentz peak with a Gaussian peak.

The formula used to calculate this is:

\[
z(x) = \frac{x + 1j \gamma}{\sqrt{2} \sigma}
\]

\[
w(z) = \exp(-z^2) \text{erfc}(-1j z) / (\sqrt{2 \pi} \sigma)
\]

\[
V(x) = \text{scale} \ Re(w(z(x-center)))
\]

Parameters

• gamma (real) – The half-width half-maximum of the Lorentzian
• FWHM (real) – The FWHM of the Gaussian
• center (real) – Location of the center of the peak
• scale (real) – Value at the highest point of the peak
Notes

adjusted to use stddev and HWHM rather than FWHM parameters

`hyperspy._components.volume_plasmon_drude` module

class `hyperspy._components.volume_plasmon_drude.VolumePlasmonDrude`:

```python
intensity=1.0, plasmon_energy=15.0, fwhm=1.5, module='numexpr',
compute_gradients=False, **kwargs)
```

Bases: `hyperspy._components.expression.Expression`

Drude volume plasmon energy loss function component, the energy loss function is defined as:

\[
f(E) = I_0 \frac{E(\Delta E_p)E_p^2}{(E^2 - E_p^2)^2 + (E\Delta E_p)^2}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_0)</td>
<td>intensity</td>
</tr>
<tr>
<td>(E_p)</td>
<td>plasmon_energy</td>
</tr>
<tr>
<td>(\Delta E_p)</td>
<td>fwhm</td>
</tr>
</tbody>
</table>

Parameters

- `intensity (float)` -
- `plasmon_energy (float)` -
- `fwhm (float)` -

Notes


`grad_fwhm(x)`
`grad_intensity(x)`
`grad_plasmon_energy(x)`

Module contents

`hyperspy._signals` package

Submodules
hyperspy._signals.common_signal1d module

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.

    See also:

    transpose(), as_signal1D(), as_signal2D(), hs.transpose()

    Raises DataDimensionError – when data.ndim < 2:

hyperspy._signals.common_signal2d module

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:

    as_signal1D() a method for the same purpose with more options.

    signals.Signal1D.to_signal1D() performs the inverse operation on one dimensional(), as_signal2D(), transpose(), hs.transpose()

hyperspy._signals.complex_signal module

class hyperspy._signals.complex_signal.ComplexSignal (*args, **kwargs)

    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 2\pi for negative values.
Parameters `deg` *(bool, optional)* – Return angle in degrees if True, radians if False (default).

Returns `angle` – The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

Return type HyperSpy signal

```python
class hyperspy._signals.complex_signal.ComplexSignal_mixin(*args, **kwargs)
Bases: object

BaseSignal subclass for complex data.
```

property `amplitude`
Get/set the amplitude of the data. Returns an appropriate HyperSpy signal.

```python
angle(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, optional)* – Return angle in degrees if True, radians if False (default).

Returns `angle` – The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

Return type HyperSpy signal

```python
change_dtype(dtype)
```
Change the data type.

Parameters `dtype` *(str or 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 imaginary part of the data. Returns an appropriate HyperSpy signal.

property `phase`
Get/set the phase of the data. Returns an appropriate HyperSpy signal.

```python
plot(power_spectrum=False, navigator='auto', axes_manager=None, representation='cartesian', norm='auto', fft_shift=False, same_axes=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 \( \text{navigation\_dimension} > 2 \), 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.

If 'slider':

• If \( \text{navigation\_dimension} > 0 \) a window with one slider per axis is raised to navigate the data.

If 'spectrum':

• If \( \text{navigation\_dimension} > 0 \) the navigator is always a spectrum obtained by integrating the data over all other axes.

If None, no navigator will be provided.

Alternatively a \texttt{BaseSignal} (or subclass) instance can be provided. The \texttt{signal\_dimension} must be 1 (for a spectrum navigator) or 2 (for a image navigator) and \texttt{navigation\_shape} must be 0 (for a static navigator) or \texttt{navigation\_shape} + \texttt{signal\_shape} must be equal to the \texttt{navigator\_shape} of the current object (for a dynamic navigator). If the signal \texttt{dtype} is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

\textbf{axes\_manager} [None or \texttt{AxesManager}] If None, the signal’s \texttt{axes\_manager} attribute is used.

\textbf{plot\_markers} [bool, default True] Plot markers added using \texttt{s.add\_marker(marker, permanent=True)}. Note, a large number of markers might lead to very slow plotting.

\textbf{norm} [str, optional] The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. (default value is 'auto'). If 'auto', intensity is plotted on a linear scale except when \texttt{power\_spectrum=True} (only for complex signals).

\textbf{power\_spectrum} [bool, default is False.] If True, plot the power spectrum instead of the actual signal, if False, plot the real and imaginary parts of the complex signal.

\textbf{representation} [['cartesian' or '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').

\textbf{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.

\textbf{fft\_shift} [bool, default False] If True, shift the zero-frequency component. See \texttt{numpy.fft.fftshift} for more details.

**kwargs Only for \texttt{Signal2D}: additional (optional) keyword arguments for \texttt{matplotlib.pyplot.imshow()}.

\textbf{property real} Get/set the real part of the data. Returns an appropriate HyperSpy signal.

\textbf{unwrapped\_phase} (\texttt{wrap\_around=False, seed=None, show\_progressbar=None, parallel=None}) Return the unwrapped phase as an appropriate HyperSpy signal.

Parameters
• **wrap_around** *(bool or sequence of bool, optional)* – 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** *(int, optional)* – 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.

• **parallel** *(None or bool)* – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns**  
**phase_image** – Unwrapped phase.

**Return type**  
BaseSignal subclass

**Notes**


class hyperspy._signals.complex_signal.LazyComplexSignal(*args, **kwargs)
Bases: hyperspy._signals.complex_signal.ComplexSignal, hyperspy._signals.lazy.LazySignal

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 $2\pi$ for negative values.

Parameters **deg** *(bool, optional)* – Return angle in degrees if True, radians if False (default).

Returns **angle** – The counterclockwise angle from the positive real axis on the complex plane, with dtype as numpy.float64.

Return type  
HyperSpy signal

hyperspy._signals.complex_signal.format_title(thing)

**hyperspy._signals.complex_signal1d module**

class hyperspy._signals.complex_signal1d.ComplexSignal1D(*args, **kwargs)
Bases: hyperspy._signals.complex_signal1d.ComplexSignal1D, hyperspy._signals.common_signal1d.CommonSignal1D

BaseSignal subclass for complex 1-dimensional data.

class hyperspy._signals.complex_signal1d.LazyComplexSignal1D(*args, **kwargs)
Bases: hyperspy._signals.complex_signal1d.ComplexSignal1D, hyperspy._signals.complex_signal1d.LazyComplexSignal

BaseSignal subclass for lazy complex 1-dimensional data.
class hyperspy._signals.complex_signal2d.Complex2Dmixin(*args, **kw)

BaseSignal subclass for complex 2-dimensional data.

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 according scale taken into account. Both are available via the axes_manager of the signal.

plot(power_spectrum=False, norm='auto', fft_shift=False, colorbar=True, scalebar=True, scalebar_color='white', axes_ticks=None, saturated_pixels=None, vmin=None, vmax=None, no_nans=False, centre_colormap='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.

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.

If `None`, no navigator will be provided.

Alternatively a `BaseSignal` (or subclass) instance can be provided. The `signal_dimension` must be 1 (for a spectrum navigator) or 2 (for a image navigator) and `navigation_shape` must be 0 (for a static navigator) or `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.

`power_spectrum` [bool, default is 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' or '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.

`colorbar` [bool, optional] If true, a colorbar is plotted for non-RGB images.

`saturated_pixels` [scalar] The percentage of pixels that are left out of the bounds. For example, the low and high bounds of a value of 1 are the 0.5% and 99.5% percentiles. It must be in the [0, 100] range. If None (default value), the value from the preferences is used.

`vmin, vmax` [scalar, optional] `vmin` and `vmax` are used to normalize luminance data.

`no_nans` [bool, optional] If True, set nans to zero for plotting.

`centre_colormap` [‘auto’, True, False] 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] 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 Only for `Signal2D`: additional (optional) keyword arguments for `matplotlib.pyplot.imshow()`.

```python
class hyperspy._signals.complex_signal2d.ComplexSignal2D(*args, **kw)
Bases: hyperspy._signals.complex_signal2d.Complex2Dmixin, hyperspy.
```
HyperSpy Documentation, Release 1.5.1.dev

_base_signal.complex_signal.ComplexSignal, hyperspy._signals.common_signal2d.CommonSignal2D

BaseSignal subclass for complex 2-dimensional data.

class hyperspy._signals.complex_signal2d.LazyComplexSignal2D(*args, **kw)

Bases: hyperspy._signals.complex_signal2d.ComplexSignal2D, hyperspy._signals.complex_signal.LazyComplexSignal

BaseSignal subclass for lazy complex 2-dimensional data.

hyperspy._signals.dielectric_function module

class hyperspy._signals.dielectric_function.DielectricFunction(*args, **kwarg)

Bases: hyperspy._signals.dielectric_function.DielectricFunction_mixin, hyperspy._signals.complex_signal1d.ComplexSignal1D

class hyperspy._signals.dielectric_function.DielectricFunction_mixin

Bases: object

get_electron_energy_loss_spectrum(zlp, t)

get_number_of_effective_electrons(nat, cumulative=False)

Compute the number of effective electrons using the Bethe f-sum rule.

The Bethe f-sum rule gives rise to two definitions of the effective number (see [Egerton2011]), neff1 and neff2:

\[ n_{eff_1} = n_{eff} \left( -\Im \left( \epsilon^{-1} \right) \right) \]

and:

\[ n_{eff_2} = n_{eff} (\epsilon_2) \]

This method computes and return both.

Parameters

- **nat** (float) – Number of atoms (or molecules) per unit volume of the sample.
- **cumulative** (bool) – If False calculate the number of effective electrons up to the higher energy-loss of the spectrum. If True, calculate the number of effective electrons as a function of the energy-loss up to the higher energy-loss of the spectrum. True is only supported by SciPy newer than 0.13.2.

Returns neff1, neff2 – Signal1D instances containing neff1 and neff2. The signal and navigation dimensions are the same as the current signal if cumulative is True, otherwise the signal dimension is 0 and the navigation dimension is the same as the current signal.

Return type Signal1D
hyperSpy documentation, release 1.5.1.dev

Notes


class hyperspy._signals.dielectric_function.LazyDielectricFunction(*args, **kwargs)
Bases: hyperspy._signals.dielectric_function.DielectricFunction, hyperspy._signals.complex_signal1d.LazyComplexSignal1D

hyperspy._signals.eds module

class hyperspy._signals.eds.EDSSpectrum(*args, **kwargs)
Bases: hyperspy._signals.eds.EDS_mixin, hyperspy._signals.signal1d.Signal1D
class hyperspy._signals.eds.EDS_mixin(*args, **kwargs)
Bases: object

add_elements(elements)
Add elements and the corresponding X-ray lines.
The list of elements is stored in metadata.Sample.elements

Parameters

- elements (list of strings) – The symbol of the elements.

Examples

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> print(s.metadata.Sample.elements)
>>> s.add_elements(['Ar'])
>>> print(s.metadata.Sample.elements)
['Al', 'C', 'Cu', 'Mn', 'Zr']
[]
```

See also:

set_elements(), add_lines(), set_lines()

add_lines(lines=(), only_one=True, only_lines=('a',))
Add X-rays lines to the internal list.

Although most functions do not require an internal list of X-ray lines because they can be calculated from
the internal list of elements, occasionally it might be useful to customize the X-ray lines to be used by all
functions by default using this method. The list of X-ray lines is stored in metadata.Sample.xray_lines

Parameters

- lines (list of strings) – A list of valid element X-ray lines to add e.g. Fe_Kb.
  Additionally, if metadata.Sample.elements is defined, add the lines of those elements that
  where not given in this list. If the list is empty (default), and metadata.Sample.elements is
  defined, add the lines of all those elements.

- only_one (bool) – If False, add all the lines of each element in metadata.Sample.elements that has not line defined in lines. If True (default), only add the
  line at the highest energy above an overvoltage of 2 (< beam energy / 2).

- only_lines ((None, list of strings)) – If not None, only the given lines
  will be added.
Examples

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.add_lines()
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_La', 'Mn_La', 'Zr_La']

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.set_microscope_parameters(beam_energy=30)
>>> s.add_lines()
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_Ka', 'Mn_Ka', 'Zr_La']

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.add_lines()
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_La', 'Mn_La', 'Zr_La']

>>> s.add_lines(['Cu_Ka'])
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_La', 'Mn_La', 'Zr_La']
```

See also:

`set_lines()`, `add_elements()`, `set_elements()`

**add_xray_lines_markers** (*xray_lines*)

Add marker on a spec.plot() with the name of the selected X-ray lines

**Parameters**

- **xray_lines** (*list of string*) – A valid list of X-ray lines

**estimate_background_windows** (*line_width=[2, 2], windows_width=1, xray_lines=None*)

Estimate two windows around each X-ray line containing only the background.

**Parameters**

- **line_width** (*list of two floats*) – The position of the two windows around the X-ray line is given by the *line_width* (left and right) times the calculated FWHM of the line.

- **windows_width** (*float*) – The width of the windows is is the *windows_width* times the calculated FWHM of the line.

- **xray_lines** (*None or list of string*) – If None, use `metadata.Sample.elements.xray_lines`. Else, provide an iterable containing a list of valid X-ray lines symbols.

**Returns**

- **windows_position** – The position of the windows in energy. Each line corresponds to a X-ray line. In a line, the two first values correspond to the limits of the left window and the two last values correspond to the limits of the right window.

**Return type**

2D array of float

Examples

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> bw = s.estimate_background_windows(line_width=[5.0, 2.0])
>>> s.plot(background_windows=bw)
```
>>> s.get_lines_intensity(background_windows=bw, plot_result=True)
Fe_Ka at 6.4039 keV : Intensity = 2754.00
Pt_La at 9.4421 keV : Intensity = 15090.00

See also:
plot(), get_lines_intensity()

estimate_integration_windows (windows_width=2.0, xray_lines=None)
Estimate a window of integration for each X-ray line.

Parameters

• windows_width (float) – The width of the integration windows is the ‘windows_width’ times the calculated FWHM of the line.

• xray_lines (None or list of string) – If None, use ‘metadata.Sample.elements.xray_lines’. Else, provide an iterable containing a list of valid X-ray lines symbols.

Returns integration_windows – The positions of the windows in energy. Each row corresponds to a X-ray line. Each row contains the left and right value of the window.

Return type 2D array of float

Examples

>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> iw = s.estimate_integration_windows()
>>> s.plot(integration_windows=iw)
Fe_Ka at 6.4039 keV : Intensity = 3710.00
Pt_La at 9.4421 keV : Intensity = 15872.00

See also:
plot(), get_lines_intensity()

get_lines_intensity (xray_lines=None, integration_windows=2.0, background_windows=None, plot_result=False, only_one=True, only_lines=('a',), **kwargs)
Return the intensity map of selected X-ray lines.

The intensities, the number of X-ray counts, are computed by suming the spectrum over the different X-ray lines. The sum window width is calculated from the energy resolution of the detector as defined in ‘energy_resolution_MnKa’ of the metadata. Backgrounds average in provided windows can be subtracted from the intensities.

Parameters

• xray_lines ((None, list of string)) – If None, if metadata.Sample.elements.xray_lines contains a list of lines use those. If metadata.Sample.elements.xray_lines is undefined or empty but metadata.Sample.elements is defined, use the same syntax as add_line to select a subset of lines for the operation. Alternatively, provide an iterable containing a list of valid X-ray lines symbols.

• integration_windows (Float or array) – If float, the width of the integration windows is the ‘integration_windows_width’ times the calculated FWHM of the line. Else
provide an array for which each row corresponds to a X-ray line. Each row contains the left and right value of the window.

- **background_windows** *(None or 2D array of float)* – If None, no background subtraction. Else, the backgrounds average in the windows are subtracted from the return intensities. ‘background_windows’ provides the position of the windows in energy. Each line corresponds to a X-ray line. In a line, the two first values correspond to the limits of the left window and the two last values correspond to the limits of the right window.

- **plot_result** *(bool)* – If True, plot the calculated line intensities. If the current object is a single spectrum it prints the result instead.

- **only_one** *(bool)* – If False, use all the lines of each element in the data spectral range. If True use only the line at the highest energy above an overvoltage of 2 (< beam energy / 2).

- **only_lines** *(None, list of strings)* – If not None, use only the given lines.

- **kwargs** – The extra keyword arguments for plotting. See **utils.plot.plot_signals**

Returns **intensities** – A list containing the intensities as BaseSignal subclasses.

Return type **list**

**Examples**

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.get_lines_intensity(['Mn_Ka'], plot_result=True)
Mn_La at 0.63316 keV : Intensity = 96700.00

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot(['Mn_Ka'], integration_windows=2.1)
>>> s.get_lines_intensity(['Mn_Ka'],
                          integration_windows=2.1, plot_result=True)
Mn_Ka at 5.8987 keV : Intensity = 53597.00

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.set_elements(['Mn'])
>>> s.set_lines(['Mn_Ka'])
>>> bw = s.estimate_background_windows()
>>> s.plot(background_windows=bw)
>>> s.get_lines_intensity(background_windows=bw, plot_result=True)
Mn_Ka at 5.8987 keV : Intensity = 46716.00
```

See also:

* set_elements(), add_elements(), estimate_background_windows(), plot() *

**get_take_off_angle()**

Calculate the take-off-angle (TOA).

TOA is the angle with which the X-rays leave the surface towards the detector. Parameters are read in ‘SEM.Stage.tilt_alpha’, ‘Acquisition_instrument.SEM.Detector.EDS.azimuth_angle’ and ‘SEM.Detector.EDS.elevation_angle’ in ‘metadata’.

Returns **take_off_angle** – in Degree

Return type **float**
Examples

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
``` 

```python
>>> s.get_take_off_angle()
37.0
``` 

```python
>>> s.set_microscope_parameters(tilt_stage=20.)
``` 

```python
>>> s.get_take_off_angle()
57.0
``` 

See also:

- `hs.eds.take_off_angle()`

Notes

Defined by M. Schaffer et al., Ultramicroscopy 107(8), pp 587-597 (2007)

```python
plot (xray_lines=False, only_lines=('a', 'b'), only_one=False, background_windows=None, integration_windows=None, **kwargs)
``` 

Plot the EDS spectrum. The following markers can be added

- The position of the X-ray lines and their names.
- The background windows associated with each X-ray lines. A black line links the left and right window with the average value in each window.

Parameters

- `xray_lines` (False, True, 'from_elements', list of string) – If not False, indicate the position and the name of the X-ray lines. If True, if `metadata.Sample.elements.xray_lines` contains a list of lines use those. If `metadata.Sample.elements.xray_lines` is undefined or empty or if `xray_lines` equals 'from_elements' and `metadata.Sample.elements` is defined, use the same syntax as `add_line` to select a subset of lines for the operation. Alternatively, provide an iterable containing a list of valid X-ray lines symbols.

- `only_lines` (None or list of strings) – If not None, use only the given lines (eg. ('a','Kb')). If None, use all lines.

- `only_one` (bool) – If False, use all the lines of each element in the data spectral range. If True use only the line at the highest energy above an overvoltage of 2 (< beam energy / 2).

- `background_windows` (None or 2D array of float) – If not None, add markers at the position of the windows in energy. Each line corresponds to a X-ray lines. In a line, the two first value corresponds to the limit of the left window and the two last values corresponds to the limit of the right window.

- `integration_windows` (None or 'auto' or float or 2D array of float) – If not None, add markers at the position of the integration windows. If 'auto' (or float), the width of the integration windows is 2.0 (or float) times the calculated FWHM of the line. see 'estimate_integration_windows'. Else provide an array for which each row corresponds to a X-ray line. Each row contains the left and right value of the window.

- `navigator` (str, None, or `BaseSignal` (or subclass)) – 

- string values are 'auto', 'slider', and 'spectrum'. (Allowed)
• 'auto' (If) –
  – 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.

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.

If None, no navigator will be provided.

Alternatively a BaseSignal (or subclass) instance can be provided. The signal_dimension
must be 1 (for a spectrum navigator) or 2 (for a image navigator) and
navigation_shape must be 0 (for a static navigator) or 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.

• norm (str, optional) – The function used to normalize the data prior to plotting.
  Allowable strings are: 'auto', 'linear', 'log'. (default value is 'auto'). If
  'auto', intensity is plotted on a linear scale except when power_spectrum=True
  (only for complex signals).

Examples

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot()

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot(True)
```
HyperSpy Documentation, Release 1.5.1.dev

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> bw = s.estimate_background_windows()
>>> s.plot(background_windows=bw)
```

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.plot(['Mn_Ka'], integration_windows='auto')
```

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> bw = s.estimate_background_windows()
>>> s.plot(background_windows=bw, integration_windows=2.1)
```

See also:

set_elements(), add_elements(), estimate_integration_windows(),
get_lines_intensity(), estimate_background_windows()

`rebin(new_shape=None, scale=None, crop=True, out=None)`
Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify either `new_shape` or `scale`.

Parameters

- **new_shape** *(list (of floats or integer) or None)* – For each dimension specify the new_shape. This will internally be converted into a `scale` parameter.

- **scale** *(list (of floats or integer) 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.
  - e.g. 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.

- **out** *(BaseSignal (or subclasses) 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 *s* – The resulting cropped signal.

Return type `BaseSignal` (or subclass)

Examples

3.1. hyperspy package
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDSTEMSpectrum, 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)
<EDSTEMSpectrum, title: dimensions (2, 2|2)>
>>> print('Sum = ', sum(sum(sum(test.data))))
Sum = 164.0

**remove_xray_lines_markers** *(xray_lines)*
Remove marker previously added on a spec.plot() with the name of the selected X-ray lines

**Parameters**  
**xray_lines** *(list of string)* – A valid list of X-ray lines to remove

**set_elements** *(elements)*
Erase all elements and set them.

**Parameters**  
**elements** *(list of strings)* – A list of chemical element symbols.

See also:

*add_elements(), set_lines(), add_lines()*

**Examples**

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> print(s.metadata.Sample.elements)
['Al', 'C', 'Cu', 'Mn', 'Zr']
>>> s.set_elements(['Al'])
>>> print(s.metadata.Sample.elements)
['Al']
```

**set_lines** *(lines, only_one=True, only_lines=('a',))*
Erase all Xrays lines and set them.

See add_lines for details.

**Parameters**

- **lines** *(list of strings)* – A list of valid element X-ray lines to add e.g. Fe_Kb. Additionally, if *metadata.Sample.elements* is defined, add the lines of those elements that where not given in this list.

- **only_one** *(bool)* – If False, add all the lines of each element in *metadata.Sample.elements* that has not line defined in lines. If True (default), only add the line at the highest energy above an overvoltage of 2 (< beam energy / 2).

- **only_lines** *(None, list of strings)* – If not None, only the given lines will be added.

**Examples**
```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.add_lines()
>>> print(s.metadata.Sample.xray_lines)
['Cu_Ka']

>>> s.set_lines(['Cu_Ka'])
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_Ka', 'Mn_La', 'Zr_La']

See also:
add_lines(), add_elements(), set_elements()

sum(axis=None, out=None)

Sum the data over the given axes.

Parameters

- **axis** (int, str, DataAxis, tuple (of DataAxis) or None) – 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns

- **s** – A new Signal containing the sum of the provided Signal along the specified axes.

Return type

BaseSignal (or subclasses)

See also:
max(), min(), mean(), std(), var(), indexmax(), indexmin(), valuemax(), valuemin()

Examples

```}

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.sum(-1).data.shape
(64,64)
```
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 strings 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.

\[
\text{mapping} = \{\text{'Acquisition_instrument.SEM.Detector.EDS.azimuth_angle': 'azimuth_angle', ...
\}
\]

```python
class hyperspy._signals.eds_sem.EDSSEMSpectrum(*args, **kwargs)

Bases: hyperspy._signals.eds_sem.EDSEM_mixin, hyperspy._signals.eds.EDSSpectrum

create_model(auto_background=True, auto_add_lines=True, *args, **kwargs)

Create a model for the current SEM EDS data.

Parameters

- **auto_background**(boolean, default True) – If True, adds automatically a polynomial order 6 to the model, using the edsmodel.add_polynomial_background method.

- **auto_add_lines**(boolean, default True) – If True, automatically add Gaussians for all X-rays generated in the energy range by an element using the edsmodel.add_family_lines method.

- **dictionary**(None, dict, optional) – A dictionary to be used to recreate a model. Usually generated using hyperspy.model.as_dictionary()

Returns model

Return type EDSSEMModel instance.
```

```python
class hyperspy._signals.eds_sem.EDSEM_mixin(*args, **kwargs)

Bases: object
```

```python
def get_calibration_from(ref, nb_pix=1)

Copy the calibration and all metadata of a reference.

Primary use: To add a calibration to ripple file from INCA software

Parameters

- **ref**(signal) – The reference contains the calibration in its metadata

- **nb_pix**(int) – The live time (real time corrected from the “dead time”) is divided by the number of pixel (spectrums), giving an average live time.

Examples

```python
>>> ref = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s = hs.signals.EDSSEMSpectrum( hs.datasets.example_signals.EDS_SEM_Spectrum().data)
>>> print(s.axes_manager[0].scale)
>>> s.get_calibration_from(ref)
>>> print(s.axes_manager[0].scale)
1.0
0.01
```
```
set_microscope_parameters(beam_energy=None, live_time=None, tilt_stage=None, azimuth_angle=None, elevation_angle=None, energy_resolution_MnKa=None, display=True, toolkit=None)
```

Set the microscope parameters.

If no arguments are given, raises an interactive mode to fill the values.

**Parameters**

- **beam_energy** *(float)* – The energy of the electron beam in keV
- **live_time** *(float)* – In second
- **tilt_stage** *(float)* – In degree
- **azimuth_angle** *(float)* – In degree
- **elevation_angle** *(float)* – In degree
- **energy_resolution_MnKa** *(float)* – In eV
- **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 strings 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.

**Examples**

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> print('Default value %s eV' % s.metadata.Acquisition_instrument.SEM.Detector.EDS.energy_resolution_MnKa)
>>> s.set_microscope_parameters(energy_resolution_MnKa=135.)
>>> print('Now set to %s eV' % s.metadata.Acquisition_instrument.SEM.Detector.EDS.energy_resolution_MnKa)
Default value 130.0 eV
Now set to 135.0 eV
```

**class hyperspy._signals.eds_sem.LazyEDSSEMSpectrum(*args, **kwargs)**

**Bases:** hyperspy._signals.eds_sem.EDSSEMSpectrum, hyperspy._signals.eds.LazyEDSSpectrum

**hyperspy._signals.eds_tem module**

**class hyperspy._signals.eds_tem.EDSTEMParametersUI(signal)**

**Bases:** hyperspy.signal.BaseSetMetadataItems

**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 strings 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.

mapping = {'Acquisition_instrument.TEM.Detector.EDS.azimuth_angle': 'azimuth_angle', ...

class hyperspy._signals.eds_tem.EDSTEMSpectrum(*args, **kwards)
Bases: hyperspy._signals.eds_tem.EDSTEM_mixin, hyperspy._signals.eds.EDSSpectrum

class hyperspy._signals.eds_tem.EDSTEM_mixin(*args, **kwards)
Bases: object

create_model (auto_background=True, auto_add_lines=True, *args, **kwargs)
Create a model for the current TEM EDS data.

Parameters

- auto_background (boolean, default True) – If True, adds automatically a polynomial order 6 to the model, using the edsmodel.add_polynomial_background method.

- auto_add_lines (boolean, default True) – If True, automatically add Gaussians for all X-rays generated in the energy range by an element using the edsmodel.add_family_lines method.

- dictionary ((None, dict), optional) – A dictionary to be used to recreate a model. Usually generated using hyperspy.model.as_dictionary()

Returns model

Return type EDSTEMModel instance.

decomposition (normalize_poissonian_noise=True, navigation_mask=1.0, closing=True, *args, **kwargs)
Decomposition with a choice of algorithms

The results are stored in self.learning_results

Parameters

- normalize_poissonian_noise (bool) – If True, scale the SI to normalize Poissonian noise

- navigation_mask (None or float or boolean numpy array) – The navigation locations marked as True are not used in the decomposition. If float is given the vacuum_mask method is used to generate a mask with the float value as threshold.

- closing (bool) – If true, applied a morphologic closing to the maks obtained by vacuum_mask.

- algorithm ('svd' | 'fast_svd' | 'mlpca' | 'fast_mlpca' | 'nmf' |) – 'sparse_pca' | 'mini_batch_sparse_pca'

- output_dimension (None or int) – number of components to keep/calculate

- centre (None | 'variables' | 'trials') – If None no centring is applied. If ‘variable’ the centring will be performed in the variable axis. If ‘trials’, the centring will be performed in the ‘trials’ axis. It only has effect when using the svd or fast_svd algorithms

- auto_transpose (bool) – If True, automatically transposes the data to boost performance. Only has effect when using the svd of fast_svd algorithms.
• **signal_mask** *(boolean numpy array)* – The signal locations marked as True are not used in the decomposition.

• **var_array** *(numpy array)* – Array of variance for the maximum likelihood PCA algorithm

• **var_func** *(function or numpy array)* – If function, it will apply it to the dataset to obtain the var_array. Alternatively, it can a an array with the coefficients of a polynomial.

• **polyfit** –

• **reproject** *(None | signal | navigation | both)* – If not None, the results of the decomposition will be projected in the selected masked area.

**Examples**

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> si = hs.stack([s]*3)
>>> si.change_dtype(float)
>>> si.decomposition()
```

See also:

`vacuum_mask()`

**get_calibration_from**(ref, nb_pix=1)
Copy the calibration and all metadata of a reference.

Primary use: To add a calibration to ripple file from INCA software

**Parameters**

• **ref**(signal) – The reference contains the calibration in its metadata

• **nb_pix**(int) – The live time (real time corrected from the “dead time”) is divided by the number of pixel (spectrums), giving an average live time.

**Examples**

```python
>>> ref = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s = hs.signals.EDSTEMSpectrum(  
    hs.datasets.example_signals.EDS_TEM_Spectrum().data)
>>> print(s.axes_manager[0].scale)
>>> s.get_calibration_from(ref)
>>> print(s.axes_manager[0].scale)
1.0
0.020028
```

**quantification**(intensities, method, factors, composition_units=’atomic’, navigation_mask=1.0, closing=True, plot_result=False, **kwargs)
Quantification using Cliff-Lorimer, the zeta-factor method, or ionization cross sections.

**Parameters**

• **intensities**(list of signal) – the intensity for each X-ray lines.

• **method** (*'CL' or 'zeta' or 'cross_section'*) – Set the quantification method: Cliff-Lorimer, zeta-factor, or ionization cross sections.
• **factors** *(list of float)* – The list of kfactors, zeta-factors or cross sections in same order as intensities. Note that intensities provided by Hyperspy are sorted by the alphabetical order of the X-ray lines. eg. factors = [0.982, 1.32, 1.60] for ['Al_Ka', 'Cr_Ka', 'Ni_Ka'].

• **composition_units** *(weight or atomic)* – The quantification returns the composition in atomic percent by default, but can also return weight percent if specified.

• **navigation_mask** *(None or float or signal)* – The navigation locations marked as True are not used in the quantification. If int is given the vacuum_mask method is used to generate a mask with the int value as threshold. Else provides a signal with the navigation shape.

• **closing** *(bool)* – If true, applied a morphologic closing to the mask obtained by vacuum_mask.

• **plot_result** *(bool)* – If True, plot the calculated composition. If the current object is a single spectrum it prints the result instead.

• **kwargs** – The extra keyword arguments are passed to plot.

Returns

• A list of quantified elemental maps (signal) giving the composition of the sample in weight or atomic percent.

• If the method is 'zeta' this function also returns the mass thickness profile for the data.

• If the method is 'cross_section' this function also returns the atom counts for each element.

Examples

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> kfactors = [1.450226, 5.075602] # For Fe Ka and Pt La
>>> bw = s.estimate_background_windows(line_width=[5.0, 2.0])
>>> s.plot(background_windows=bw)
>>> intensities = s.get_lines_intensity(background_windows=bw)
>>> res = s.quantification(intensities, kfactors, plot_result=True,
... composition_units='atomic')
Fe (Fe_Ka): Composition = 15.41 atomic percent
Pt (Pt_La): Composition = 84.59 atomic percent
```

See also:

vacuum_mask()

**set_microscope_parameters** *(beam_energy=None, live_time=None, tilt_stage=None, azimuth_angle=None, elevation_angle=None, energy_resolution_MnKa=None, beam_current=None, probe_area=None, real_time=None, display=True, toolkit=None)*

Set the microscope parameters.

If no arguments are given, raises an interactive mode to fill the values.

Parameters
- **beam_energy** *(float)* – The energy of the electron beam in keV
- **live_time** *(float)* – In seconds
- **tilt_stage** *(float)* – In degree
- **azimuth_angle** *(float)* – In degree
- **azimuth_angle** *(float)* – In degree
- **energy_resolution_MnKa** *(float)* – In eV
- **beam_current** *(float)* – In nA
- **probe_area** *(float)* – In nm²
- **real_time** *(float)* – In seconds
- **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 strings 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.

### Examples

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> print(s.metadata.Acquisition_instrument.
>>>     TEM.Detector.EDS.energy_resolution_MnKa)
>>> s.set_microscope_parameters(energy_resolution_MnKa=135.)
>>> print(s.metadata.Acquisition_instrument.
>>>     TEM.Detector.EDS.energy_resolution_MnKa)
133.312296
135.0
```

#### vacuum_mask *(threshold=1.0, closing=True, opening=False)*

Generate mask of the vacuum region

**Parameters**

- **threshold** *(float)* – For a given pixel, maximum value in the energy axis below which the pixel is considered as vacuum.
- **closing** *(bool)* – If true, applied a morphologic closing to the mask
- **opening** *(bool)* – If true, applied a morphologic opening to the mask

### Examples

```python
>>> # Simulate a spectrum image with vacuum region
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s_vac = hs.signals.BaseSignal(np.ones_like(s.data, dtype=float))*0.005
>>> s_vac.add_poissonian_noise()
>>> si = hs.stack([s]*3 + [s_vac])
>>> si.vacuum_mask().data
array([[False, False, False, True],
       [False, False, False, False]])
```
Returns mask – The mask of the region

Return type signal

class hyperspy._signals.eds_tem.LazyEDSTEMSpectrum(*args, **kwargs)
    Bases: hyperspy._signals.eds_tem.EDSTEMSpectrum, hyperspy._signals.eds.LazyEDSSpectrum

hyperspy._signals.eels module

class hyperspy._signals.eels.EELSSpectrum(*args, **kwargs)
    Bases: hyperspy._signals.eels.EELSSpectrum_mixin, hyperspy._signals.signal1d.Signal1D

class hyperspy._signals.eels.EELSSpectrum_mixin(*args, **kwargs)
    Bases: object

    add_elements(elements, include_pre_edges=False)
        Declare the elemental composition of the sample.
        The ionisation edges of the elements present in the current energy range will be added automatically.

        Parameters
        • elements (tuple of strings) – The symbol of the elements. Note this input must always be in the form of a tuple. Meaning: add_elements(('C',)) will work, while add_elements(('C')) will NOT work.
        • include_pre_edges (bool) – If True, the ionization edges with an onset below the lower energy limit of the SI will be included

    Examples

    >>> s = hs.signals.EELSSpectrum(np.arange(1024))
    >>> s.add_elements(('C', 'O'))

    Raises ValueError –

    align_zero_loss_peak(calibrate=True, also_align=[], print_stats=True, subpixel=True, mask=None, signal_range=None, show_progressbar=None, crop=True, **kwargs)
        Align the zero-loss peak.
        This function first aligns the spectra using the result of estimate_zero_loss_peak_centre and afterward, if subpixel is True, proceeds to align with subpixel accuracy using align1D. The offset is automatically correct if calibrate is True.

        Parameters
        • calibrate (bool) – If True, set the offset of the spectral axis so that the zero-loss peak is at position zero.
        • also_align (list of signals) – A list containing other spectra of identical dimensions to align using the shifts applied to the current spectrum. If calibrate is True, the calibration is also applied to the spectra in the list.
        • print_stats (bool) – If True, print summary statistics of the ZLP maximum before the alignment.
• **subpixel** *(bool)* – If True, perform the alignment with subpixel accuracy using cross-correlation.

• **mask** *(Signal1D of bool data type.)* – 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.

• **signal_range** *(tuple of integers, tuple of floats. Optional)* – Will only search for the ZLP within the signal_range. If given in integers, the range will be in index values. If given floats, the range will be in spectrum values. Useful if there are features in the spectrum which are more intense than the ZLP. Default is searching in the whole signal.

• **show_progressbar** *(None or bool)* – If True, display a progress bar. If None, the default from the preferences settings is used.

• **crop** *(bool)* – If True automatically crop the signal axis at both ends if needed.

### Examples

```python
>>> s_ll = hs.signals.EELSSpectrum(np.zeros(1000))
>>> s_ll.data[100] = 100
>>> s_ll.align_zero_loss_peak()
```

Aligning both the lowloss signal and another signal

```python
>>> s = hs.signals.EELSSpectrum(np.range(1000))
>>> s_ll.align_zero_loss_peak(also_align=[s])
```

Aligning within a narrow range of the lowloss signal

```python
>>> s_ll.align_zero_loss_peak(signal_range=(-10.,10.))
```

See also:

`estimate_zero_loss_peak_centre()`, `align1D()`, `estimate_shift1D()`.

### Notes

Any extra keyword arguments are passed to `align1D`. For more information read its docstring.

**create_model**(ll=None, auto_background=True, auto_add_edges=True, GOS=None, dictionary=None)

Create a model for the current EELS data.

**Parameters**

• **ll** *(EELSSpectrum, optional)* – If an EELSSpectrum is provided, it will be assumed that it is a low-loss EELS spectrum, and it will be used to simulate the effect of multiple scattering by convolving it with the EELS spectrum.

• **auto_background** *(boolean, default True)* – If True, and if spectrum is an EELS instance adds automatically a powerlaw to the model and estimate the parameters by the two-area method.

• **auto_add_edges** *(boolean, default True)* – If True, and if spectrum is an EELS instance, it will automatically add the ionization edges as defined in
the Signal1D instance. Adding a new element to the spectrum using the components.EELSSpectrum.add_elements method automatically adds the corresponding ionisation edges to the model.

- **GOS** ("hydrogenic" | 'Hartree-Slater'), optional – The generalized oscillation strength calculations to use for the core-loss EELS edges. If None the Hartree-Slater GOS are used if available, otherwise it uses the hydrogenic GOS.

- **dictionary**(None | dict), optional – A dictionary to be used to recreate a model. Usually generated using hyperspy.model.as_dictionary()

**Returns** model

**Return type** EELSModel instance.

**estimate_elastic_scattering_intensity** (threshold, show_progressbar=None)

Rough estimation of the elastic scattering intensity by truncation of a EELS low-loss spectrum.

**Parameters**

- **threshold** (Signal1D, float, int) – Truncation energy to estimate the intensity of the elastic scattering. The threshold can be provided as a signal of the same dimension as the input spectrum navigation space containing the threshold value in the energy units. Alternatively a constant threshold can be specified in energy/index units by passing float/int.

- **show_progressbar** (None or bool) – If True, display a progress bar. If None, the default from the preferences settings is used.

**Returns** I0 – The elastic scattering intensity.

**Return type** Signal1D

See also:

**estimate_elastic_scattering_threshold()**

**estimate_elastic_scattering_threshold** (window=10.0, tol=None, window_length=5, polynomial_order=3, start=1.0)

Calculate the first inflexion point of the spectrum derivative within a window.

This method assumes that the zero-loss peak is located at position zero in all the spectra. Currently it looks for an inflexion point, that can be a local maximum or minimum. Therefore, to estimate the elastic scattering threshold start + window must be less than the first maximum for all spectra (often the bulk plasmon maximum). If there is more than one inflexion point in energy the window it selects the smoother one what, often, but not always, is a good choice in this case.

**Parameters**

- **window** (None, float) – If None, the search for the local inflexion point is performed using the full energy range. A positive float will restrict the search to the (0,window] energy window, where window is given in the axis units. If no inflexion point is found in this spectral range the window value is returned instead.

- **tol** (None, float) – The threshold tolerance for the derivative. If “auto” it is automatically calculated as the minimum value that guarantees finding an inflexion point in all the spectra in given energy range.

- **window_length** (int) – If non zero performs order three Savitzky-Golay smoothing to the data to avoid falling in local minima caused by the noise. It must be an odd integer.

- **polynomial_order** (int) – Savitzky-Golay filter polynomial order.
• **start** *(float)* – Position from the zero-loss peak centre from where to start looking for
the inflexion point.

**Returns** threshold – A Signal1D of the same dimension as the input spectrum navigation space
containing the estimated threshold. Where the threshold couldn’t be estimated the value is
set to nan.

**Return type** Signal1D

See also:

*estimate_elastic_scattering_intensity()*,
*align_zero_loss_peak()*,
*find_peaks1D_ohaver()*,
*fourier_ratio_deconvolution()*

**Notes**

The main purpose of this method is to be used as input for *estimate_elastic_scattering_intensity*. Indeed,
for currently achievable energy resolutions, there is not such a thing as an elastic scattering threshold.
Therefore, please be aware of the limitations of this method when using it.

**estimate_thickness** *(threshold, zlp=None)*

Estimates the thickness (relative to the mean free path) of a sample using the log-ratio method.

The current EELS spectrum must be a low-loss spectrum containing the zero-loss peak. The hyperspectrum
must be well calibrated and aligned.

**Parameters**

• **threshold** *(Signal1D, float, int)* – Truncation energy to estimate the inten-
sity of the elastic scattering. The threshold can be provided as a signal of the same
dimension as the input spectrum navigation space containing the threshold value in the
energy units. Alternatively a constant threshold can be specified in energy/index units by
passing float/int.

• **zlp** *(None, EELSSpectrum)* – If not None the zero-loss peak intensity is calcu-
lated from the ZLP spectrum supplied by integration using Simpson’s rule. If None esti-
mates the zero-loss peak intensity using *estimate_elastic_scattering_intensity* by trunca-
tion.

**Returns** s – The thickness relative to the MFP. It returns a Signal1D, Signal2D or a BaseSignal,
depending on the current navigation dimensions.

**Return type** Signal1D

**Notes**

For details see: Egerton, R. Electron Energy-Loss Spectroscopy in the Electron Microscope. Springer-
Verlag, 2011.

**estimate_zero_loss_peak_centre** *(mask=None)*

Estimate the position of the zero-loss peak.

This function provides just a coarse estimation of the position of the zero-loss peak centre by computing
the position of the maximum of the spectra. For subpixel accuracy use *estimate_shift1D*.

**Parameters** mask *(Signal1D of bool data type.)* – 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.

**Returns** zlpc – The estimated position of the maximum of the ZLP peak.
Return type  Signal1D subclass

Notes
This function only works when the zero-loss peak is the most intense feature in the spectrum. If it is not in most cases the spectrum can be cropped to meet this criterium. Alternatively use estimate_shift1D.

See also:
estimate_shift1D(), align_zero_loss_peak()
fourier_log_deconvolution(zlp, add_zlp=False, crop=False)
Performs fourier-log deconvolution.

Parameters
• zlp (EELSSpectrum) – The corresponding zero-loss peak.
• add_zlp (bool) – If True, adds the ZLP to the deconvolved spectrum
• crop (bool) – If True crop the spectrum to leave out the channels that have been modified to decay smoothly to zero at the sides of the spectrum.

Returns
Return type  An EELSSpectrum containing the current data deconvolved.

Notes
fourier_ratio_deconvolution(ll, fwhm=None, threshold=None, extrapolate_lowloss=True, extrapolate_coreloss=True)
Performs Fourier-ratio deconvolution.
The core-loss should have the background removed. To reduce the noise amplification the result is convolved with a Gaussian function.

Parameters
• ll (EELSSpectrum) – The corresponding low-loss (ll) EELSSpectrum.
• fwhm (float or None) – Full-width half-maximum of the Gaussian function by which the result of the deconvolution is convolved. It can be used to select the final SNR and spectral resolution. If None, the FWHM of the zero-loss peak of the low-loss is estimated and used.
• threshold ([None, float]) – Truncation energy to estimate the intensity of the elastic scattering. If None the threshold is taken as the first minimum after the ZLP centre.
• extrapolate_coreloss (extrapolate_lowloss,) – If True the signals are extrapolated using a power law,
Notes


generate_subshells(include_pre_edges=False)

Calculate the subshells for the current energy range for the elements present in self.elements

Parameters

- **include_pre_edges** *(bool)* – If True, the ionization edges with an onset below the lower energy limit of the SI will be included

kramers_kronig_analysis(zlp=None, iterations=1, n=None, t=None, delta=0.5, full_output=False)

Calculate the complex dielectric function from a single scattering distribution (SSD) using the Kramers-Kronig relations.

It uses the FFT method as in [Egerton2011]. The SSD is an EELSSpectrum instance containing SSD low-loss EELS with no zero-loss peak. The internal loop is devised to approximately subtract the surface plasmon contribution supposing an unoxidized planar surface and neglecting coupling between the surfaces. This method does not account for retardation effects, instrumental broadening and surface plasmon excitation in particles.

Note that either refractive index or thickness are required. If both are None or if both are provided an exception is raised.

Parameters

- **zlp** *(None, number, Signal1D) – ZLP intensity. It is optional (can be None) if \( t \) is None and \( n \) is not None and the thickness estimation is not required. If \( t \) is not None, the ZLP is required to perform the normalization and if \( t \) is not None, the ZLP is required to calculate the thickness. If the ZLP is the same for all spectra, the integral of the ZLP can be provided as a number. Otherwise, if the ZLP intensity is not the same for all spectra, it can be provided as i) a Signal1D of the same dimensions as the current signal containing the ZLP spectra for each location ii) a BaseSignal of signal dimension 0 and navigation_dimension equal to the current signal containing the integrated ZLP intensity.*

- **iterations** *(int)* – Number of the iterations for the internal loop to remove the surface plasmon contribution. If 1 the surface plasmon contribution is not estimated and subtracted (the default is 1).

- **n** *(None, float)* – The medium refractive index. Used for normalization of the SSD to obtain the energy loss function. If given the thickness is estimated and returned. It is only required when \( t \) is None.

- **t** *(None, number, Signal1D)* – The sample thickness in nm. Used for normalization of the SSD to obtain the energy loss function. It is only required when \( n \) is None. If the thickness is the same for all spectra it can be given by a number. Otherwise, it can be provided as a BaseSignal with signal dimension 0 and navigation_dimension equal to the current signal.

- **delta** *(float)* – A small number (0.1-0.5 eV) added to the energy axis in specific steps of the calculation the surface loss correction to improve stability.

- **full_output** *(bool)* – If True, return a dictionary that contains the estimated thickness if \( t \) is None and the estimated surface plasmon excitation and the spectrum corrected from surface plasmon excitations if iterations > 1.

Returns

- **eps** *(DielectricFunction instance)* – The complex dielectric function results,
\[ \epsilon = \epsilon_1 + i \epsilon_2, \]

contained in an DielectricFunction instance.

• output (Dictionary (optional)) – A dictionary of optional outputs with the following keys:
  
  thickness The estimated thickness in nm calculated by normalization of the SSD (only when \( t \) is None)
  
  surface plasmon estimation The estimated surface plasmon excitation (only if \( \text{iterations} \) > 1.)

Raises

• ValueError – If both \( n \) and \( t \) are undefined (None).

• AttributeError – If the beam_energy or the collection semi-angle are not defined in metadata.

Notes

This method is based in Egerton’s Matlab code [Egerton2011] with some minor differences:

• The integrals are performed using the simpsons rule instead of using a summation.

• The wrap-around problem when computing the ffts is workarounded by padding the signal instead of substracting the reflected tail.

```
power_law_extrapolation (window_size=20, extrapolation_size=1024, add_noise=False, fix_neg_r=False)
```

Extrapolate the spectrum to the right using a powerlaw

Parameters

• window_size (int) – The number of channels from the right side of the spectrum that are used to estimate the power law parameters.

• extrapolation_size (int) – Size of the extrapolation in number of channels

• add_noise (bool) – If True, add poissonian noise to the extrapolated spectrum.

• fix_neg_r (bool) – If True, the negative values for the “components.PowerLaw” parameter \( r \) will be flagged and the extrapolation will be done with a constant zero-value.

Returns

Return type A new spectrum, with the extrapolation.

```
rebin (new_shape=None, scale=None, crop=True, out=None)
```

Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify either new_shape or scale.

Parameters

• new_shape (list (of floats or integer) or None) – For each dimension specify the new_shape. This will internally be converted into a scale parameter.

• scale (list (of floats or integer) 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.

  – e.g. 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.

- **out** *(BaseSignal or subclasses 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 s – The resulting cropped signal.

Return type BaseSignal (or subclass)

Examples

```py
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDSTEMSpectrum, 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)
<EDSTEMSpectrum, title: dimensions (2, 2|2)>

>>> print('Sum = ', sum(sum(sum(test.data))))
Sum = 164.0
```

**richardson_lucy_deconvolution**(psf, iterations=15, mask=None, show_progressbar=None, parallel=None)

1D Richardson-Lucy Poissonian deconvolution of the spectrum by the given kernel.

Parameters

- **iterations** *(int)* – Number of iterations of the deconvolution. Note that increasing the value will increase the noise amplification.

- **psf** *(EELSSpectrum)* – It must have the same signal dimension as the current spectrum and a spatial dimension of 0 or the same as the current spectrum.

- **show_progressbar** *(None or bool)* – If True, display a progress bar. If None, the default from the preferences settings is used.

- **parallel** *(None or bool)* – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.
Notes


```python
set_microscope_parameters(beam_energy=None, convergence_angle=None, collection_angle=None, toolkit=None, display=True)
```

Set the microscope parameters that are necessary to calculate the GOS.

- **beam_energy**: float  The energy of the electron beam in keV
- **convergence_angle**: [float] The microscope convergence semi-angle in mrad.
- **collection_angle**: [float] The collection semi-angle in mrad.
- **toolkit** [str, iterable of strings 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.
- **display** [bool] If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

```python
class hyperspy._signals.eels.EELSTEMParametersUI(signal)
    Bases: hyperspy.signal.BaseSetMetadataItems
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 strings 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.

```python
mapping = {'Acquisition_instrument.TEM.Detector.EELS.collection_angle': 'collection_angle', 'Acquisition_instrument.TEM.beam_energy': 'beam_energy', 'Acquisition_instrument.TEM.convergence_angle': 'convergence_angle'}
```

```python
class hyperspy._signals.eels.LazyEELSSpectrum(*args, **kwargs)
    Bases: hyperspy._signals.eels.EELSSpectrum, hyperspy._signals.signal1d.LazySignal1D
```

```python
class hyperspy._signals.hologram_image.HologramImage(*args, **kw)
    Bases: hyperspy._signals.signal2d.Signal2D
```

Image subclass for holograms acquired via off-axis electron holography.

```python
estimate_sideband_position(ap_cb_radius=None, sb='lower', high_cf=True, show_progressbar=False, parallel=None)
```

Estimates the position of the sideband and returns its position.

**Parameters**

- `ap_cb_radius (float, None)` – The aperture radius used to mask out the center-band.
• **sb** (*str, optional*) – Chooses which sideband is taken. ‘lower’ or ‘upper’

• **high_cf** (*bool, optional*) – If False, the highest carrier frequency allowed for the sideband location is equal to half of the Nyquist frequency (Default: True).

• **show_progressbar** (*None or bool*) – If True, display a progress bar. If None, the default from the preferences settings is used.

• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns**

**Return type** Signal1D instance of sideband positions (y, x), referred to the unshifted FFT.

**Examples**

```python
>>> import hyperspy.api as hs
>>> s = hs.datasets.example_signals.object_hologram()
>>> sb_position = s.estimate_sideband_position()
>>> sb_position.data
array([[124, 452]])
```

**estimate_sideband_size** (*sb_position, show_progressbar=False, parallel=None*)

Estimates the size of the sideband and returns its size.

**Parameters**

• **sb_position** (*class:* hyperspy.signals.BaseSignal*) – The sideband position (y, x), referred to the non-shifted FFT.

• **show_progressbar** (*boolean*) – Shows progressbar while iterating over different slices of the signal (passes the parameter to map method).

• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns**

**Return type** Signal 1D instance with sideband size, referred to the unshifted FFT.

**Examples**

```python
>>> import hyperspy.api as hs
>>> s = hs.datasets.example_signals.object_hologram()
>>> sb_position = s.estimate_sideband_position()
>>> sb_size = s.estimate_sideband_size(sb_position)
>>> sb_size.data
array([ 68.87670143])
```

**reconstruct_phase** (*reference=None, sb_size=None, sb_smoothness=None, sb_unit=None, sb='lower', sb_position=None, high_cf=True, output_shape=None, plotting=False, show_progressbar=False, store_parameters=True, parallel=None*)

Reconstruct electron holograms. Operates on multidimensional hyperspy signals. There are several usage schemes:

1. Reconstruct 1d or Nd hologram without reference
2. Reconstruct 1d or Nd hologram using single reference hologram

3. Reconstruct Nd hologram using Nd reference hologram (applies each reference to each hologram in Nd stack)

The reconstruction parameters (sb_position, sb_size, sb_smoothness) have to be 1d or to have same dimensionality as the hologram.

**Parameters**

- **sb_size** (float, ndarray, :class:`hyperspy.signals.BaseSignal`, None) – Sideband radius of the aperture in corresponding unit (see `sb_unit`). If None, the radius of the aperture is set to 1/3 of the distance between sideband and center band.
- **sb_smoothness** (float, ndarray, :class:`hyperspy.signals.BaseSignal`, None) – Smoothness of the aperture in the same unit as sb_size.
- **sb_unit** (str, None) – Unit of the two sideband parameters `sb_size` and `sb_smoothness`. Default: None - Sideband size given in pixels ‘nm’: Size and smoothness of the aperture are given in 1/nm. ‘mrad’: Size and smoothness of the aperture are given in mrad.
- **sb** (str, None) – Select which sideband is selected. ‘upper’ or ‘lower’.
- **sb_position** (tuple, :class:`hyperspy.signals.Signal1D`, None) – The sideband position (y, x), referred to the non-shifted FFT. If None, sideband is determined automatically from FFT.
- **high_cf** (bool, optional) – If False, the highest carrier frequency allowed for the sideband location is equal to half of the Nyquist frequency (Default: True).
- **output_shape** (tuple, None) – Choose a new output shape. Default is the shape of the input hologram. The output shape should not be larger than the input shape.
- **plotting** (boolean) – Shows details of the reconstruction (i.e. SB selection).
- **show_progressbar** (None or bool) – If True, display a progress bar. If None, the default from the preferences settings is used.
- **parallel** (None or bool) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.
- **store_parameters** (boolean) – Store reconstruction parameters in metadata

**Returns**

- **wave** – Reconstructed electron wave. By default object wave is divided by reference wave

**Return type** :class:`hyperspy.signals.WaveImage

**Examples**

```python
>>> import hyperspy.api as hs
>>> s = hs.datasets.example_signals.object_hologram()
>>> sb_position = s.estimate_sideband_position()
>>> sb_size = s.estimate_sideband_size(sb_position)
>>> sb_size.data
>>> wave = s.reconstruct_phase(sb_position=sb_position, sb_size=sb_size)
```
set_microscope_parameters(beam_energy=None, biprism_voltage=None, tilt_stage=None)
    Set the microscope parameters.
    
    If no arguments are given, raises an interactive mode to fill the values.
    
    Parameters
    
    • beam_energy(float) – The energy of the electron beam in keV
    • biprism_voltage(float) – In volts
    • tilt_stage(float) – In degrees
    
    Examples
    
    >>> s.set_microscope_parameters(beam_energy=300.)
    >>> print('Now set to %s keV' % s.metadata.Acquisition_instrument.TEM.beam_energy)
    Now set to 300.0 keV
    
    statistics(sb_position=None, sb='lower', high_cf=False, fringe_contrast_algorithm='statistical', apodization='hanning', single_values=True, show_progressbar=False, parallel=None)
    Calculates following statistics for off-axis electron holograms:
    1. Fringe contrast using either statistical definition or Fourier space approach (see description of fringe_contrast_algorithm parameter)
    2. Fringe sampling (in pixels)
    3. Fringe spacing (in calibrated units)
    4. Carrier frequency (in calibrated units, radians and 1/px)
    
    Parameters
    
    • sb_position (tuple, :class:`~hyperspy.signals.Signal1D, None) – The sideband position (y, x), referred to the non-shifted FFT. It has to be tuple or to have the same dimensionality as the hologram. If None, sideband is determined automatically from FFT.
    • sb(str, None) – Select which sideband is selected. ‘upper’, ‘lower’, ‘left’ or ‘right’.
    • high_cf(bool, optional) – If False, the highest carrier frequency allowed for the sideband location is equal to half of the Nyquist frequency (Default: False).
    • fringe_contrast_algorithm(str) – Select fringe contrast algorithm between:
      ‘fourier’ fringe contrast is estimated as: 2 * |I(k_0)| / <I(0)>, where I(k_0) is intensity of sideband and I(0) is the intensity of central band (FFT origin). This method delivers also reasonable estimation if interference pattern do not cover full field of view.
      ‘statistical’ fringe contrast is estimated by dividing standard deviation by mean of the hologram intensity in real space. This algorithm relays on that the fringes are regular and covering entire field of view.
      (Default: ‘statistical’)
    • apodization (str or None, optional) – Used with fringe_contrast_algorithm=’fourier’. If ‘hanning’ or ‘hamming’ apodization window will be applied in real space before FFT for estimation of fringe contrast. Apodization is typically needed to suppress striking due to sharp edges of the image, which often results in underestimation of the fringe contrast. (Default: ‘hanning’)
    • single_values(bool, optional) – If True calculates statistics only for the first navigation pixels and returns the values as single floats (Default: True)
• `show_progressbar` *(None or bool)* – If True, display a progress bar. If None, the default from the preferences settings is used.

• `parallel` *(None or bool)* – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns** Dictionary with the statistics

**Return type** `statistics_dict`

**Examples**

```python
>>> import hyperspy.api as hs
>>> s = hs.datasets.example_signals.reference_hologram()
>>> sb_position = s.estimate_sideband_position(high Cf=True)
>>> s.statistics(sb_position=sb_position)
{'Fringe spacing (nm)': 3.4860442674236256,
 'Carrier frequency (1/px)': 0.26383819985575441,
 'Carrier frequency (mrad)': 0.56475154609203482,
 'Fringe contrast': 0.071298357213623778,
 'Fringe sampling (px)': 3.7902017241882331,
 'Carrier frequency (1 / nm)': 0.28685808994016415}
```

```python
class hyperspy._signals.hologram_image.LazyHologramImage(*args, **kw)
Bases: hyperspy._signals.lazy.LazySignal, hyperspy._signals.hologram_image.HologramImage
hyperspy._signals.lazy module

class hyperspy._signals.lazy.LazySignal(data, **kwds)
Bases: hyperspy.signal.BaseSignal
```

A Lazy Signal instance that delays computation until explicitly saved (assuming storing the full result of computation in memory is not feasible)

Create a Signal from a numpy array.

**Parameters**

- `data` *(numpy.ndarray)* – The signal data. It can be an array of any dimensions.

- `axes` *(dict, optional)* – Dictionary 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.

**change_dtype** *(dtype, rechunk=True)*

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)` 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)` 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. If `True` (default), the data may be automatically rechunked before performing this operation.

**Examples**

```python
>>> 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(progressbar=True, close_file=False)`
Attempt to store the full signal in memory.

`close_file` : bool 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.

`decomposition(normalize_poissonian_noise=False, algorithm='svd', output_dimension=None, signal_mask=None, navigation_mask=None, get=<function get>, num_chunks=None, reproject=True, bounds=False, **kwargs)`
Perform Incremental (Batch) decomposition on the data, keeping `n` significant components.

**Parameters**

• `normalize_poissonian_noise` (bool) – If True, scale the SI to normalize Poissonian noise

• `algorithm` (str) – One of (`'svd'`, `'PCA'`, `'ORPCA'`, `'ONMF'`). By default `svd`, lazy SVD decomposition from dask.

• `output_dimension` (int) – the number of significant components to keep. If None, keep all (only valid for SVD)

• `get` (dask scheduler) – the dask scheduler to use for computations; default `dask.threaded.get`

• `num_chunks` (int) – 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 atleast `output_dimension` signals.

• `navigation_mask` (BaseSignal, numpy array, dask array) – The navigation locations marked as True are not used in the decomposition.
• **signal_mask** ([BaseSignal, numpy array, dask array]) – The signal locations marked as True are not used in the decomposition.

• **reproject** (bool) – Reproject data on the learnt components (factors) after learning.

• **kwargs** – passed to the partial_fit/fit functions.

**Notes**

Various algorithm parameters and their default values:

ONMF: lambda1=1, kappa=1, robust=False, store_r=False batch_size=None

ORPCA: fast=True, lambda1=None, lambda2=None, method=None, learning_rate=None, init=None, training_samples=None, momentum=None

PCA: batch_size=None, copy=True, white=False

**diff** (axis, order=1, out=None, rechunk=True)

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: \( \text{out}[n] = a[n+1] - a[n] \). 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

**Returns** s – 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.

**Return type** BaseSignal (or subclasses) or None

See also:

derivative(), integrate1D(), integrate_simpson()
Parameters

- **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
  - 'scotts': use Scott’s rule to determine bins
  - 'freedman': use the Freedman-diaconis rule to determine bins
  - 'blocks': use bayesian blocks for dynamic bin widths
- **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.
- **out**: (BaseSignal (or subclasses) 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. If True (default), 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** – A 1D spectrum instance containing the histogram.

Return type **Signal1D**

See also:

print_summary_statistics(), astroML.density_estimation.histogram(), numpy.histogram()

Notes

- The lazy version of the algorithm does not support the 'knuth' and 'blocks' bins arguments.
- The estimators for bins are taken from the AstroML project. Read the documentation of astroML.density_estimation.histogram() for more info.

Examples

```python
>>> 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)

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 subclasses) 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**  
s – A new Signal containing the integral of the provided Signal along the specified axis.

**Return type**  
BaseSignal (or subclasses)

See also:

diff(), derivative(), integrate1D()

**Examples**

```python
>>> import numpy as np

>>> s = BaseSignal(np.random.random((64, 64, 1024))
>>> s.data.shape
(64, 64, 1024)
>>> s.integrate_simpson(-1).data.shape
(64, 64)
```

**rebin**(new_shape=None, scale=None, crop=False, out=None, rechunk=True)

Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify either new_shape or scale.

**Parameters**

- **new_shape** (list (of floats or integer) or None) – For each dimension specify the new_shape. This will internally be converted into a scale parameter.

- **scale** (list (of floats or integer) 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.

  e.g. 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.

- **out** (BaseSignal (or subclasses) 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**  
s – The resulting cropped signal.

**Return type**  
BaseSignal (or subclass)

**Examples**

```python
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones((4, 4, 10)))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
```
<EDXTEM Spectrum, 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)
<EDXTEM Spectrum, title: dimensions (2, 2|2)>
>>> print('Sum = ', sum(sum(sum(test.data))))
Sum = 164.0

valuemax (axis, out=None, rechunk=True)
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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns s – A new Signal containing the calibrated coordinate values of the maximum along the specified axis.

Return type BaseSignal (or subclasses)

See also: max(), min(), sum(), mean(), std(), var(), indexmax(), indexmin(), valuemin()

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64, 64, 1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.valuemax(-1).data.shape
(64, 64)
```

valuemin (axis, out=None, rechunk=True)
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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.
Returns s – A new Signal containing the calibrated coordinate values of the minimum along the specified axis.

Return type BaseSignal (or subclasses)

See also: max(), min(), sum(), mean(), std(), var(), indexmax(), indexmin(), valuemax()

docs/hyperspy._signals.lazy.to_array(thing, chunks=None)

Accepts BaseSignal, dask or numpy arrays and always produces either numpy or dask array.

Parameters

- thing ({BaseSignal, dask.array.Array, numpy.ndarray}) – the thing to be converted
- chunks ({None, tuple of tuples}) – If None, the returned value is a numpy array. Otherwise returns dask array with the chunks as specified.

Returns res

Return type {numpy.ndarray, dask.array.Array}

docs/hyperspy._signals.signal1d module

class hyperspy._signals.signal1d.LazySignal1D(*args, **kwargs)

Bases: hyperspy._signals.lazy.LazySignal, hyperspy._signals.signal1d.Signal1D

class hyperspy._signals.signal1d.Signal1D(*args, **kwargs)

Bases: hyperspy.signal.BaseSignal, hyperspy._signals.common_signal1d.CommonSignal1D

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)

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

- end (start,) – 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 ints 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. Over-rides *crop*.

• **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 signals*, *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* data type.) – 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

**Return type**  An array with the result of the estimation.

**Raises**  *SignalDimensionError* – If the signal dimension is not 1.

See also:

None()

**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 offset, units and scale directly

• 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 strings 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.

Notes

For this method to work the output_dimension must be 1.

**Raises**  *SignalDimensionError* – If the signal dimension is not 1.

**create_model** (*dictionary=None*)

Create a model for the current data.

Returns  *model*

**Return type**  *Model1D* instance.

**crop_signal1D** (*args, **kwargs*)

Crop in place the spectral dimension.
Parameters **right_value**(left_value,) – 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, parallel=None, show_progressbar=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 substracted. The estimation is performed by interpolation using cubic splines.

Parameters

- **factor**(0 < float < 1) – 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.
- **parallel**(None or bool) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

Returns

- 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, parallel=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

- **end**(start,) – 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 ints 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
• **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.

• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns**

**Return type** 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.

**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 strings 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.

**find_peaks1D_ohaver** (*xdim=None, slope_thresh=0, amp_thresh=None, subchannel=True, medfilt_radius=5, maxpeakn=30000, peakgroup=10, parallel=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.

*peakgroup* is the number of points around the top of the peak that are 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.

**Parameters**

• **slope_thresh** (*float, optional*) – 1st derivative threshold to count the peak; higher values will neglect broader features; default is set to 0.

• **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, optional*) – median filter window to apply to smooth the data (see scipy.signal.medfilt); if 0, no filter will be applied; default is set to 5.

• **peakgroup** (*int, optional*) – number of points around the “top part” of the peak that are taken to estimate the peak height; default is set to 10

• **maxpeakn** (*int, optional*) – number of maximum detectable peaks; default is set to 5000.

• **subchannel** (*bool, optional*) – default is set to True.
• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Returns**

• **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.

**hanning_taper** (*side='both', channels=None, offset=0*)

Apply a hanning taper to the data in place.

**Parameters**

• **side** (*'left', 'right' or '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**

**Return type** channels

**Raises** *SignalDimensionError* – If the signal dimension is not 1.

**integrate_in_range** (*signal_range='interactive', display=True, toolkit=None*)

Sums the spectrum over an energy range, giving the integrated area. The energy range can either be selected through a GUI or the command line.

**Parameters**

* **signal_range** (*a tuple of this form (l, r) or "interactive")* – l and r are the left and right limits of the range. They can be numbers or None, where None indicates the extremes of the interval. If l and r are floats the *signal_range* will be in axis units (for example eV). If l and r are integers the *signal_range* will be in index units. When *signal_range* is “interactive” (default) the range is selected using a GUI.

**Returns** integrated_spectrum

**Return type** BaseSignal subclass

**See also:** None()

**Examples**

Using the GUI
```python
>>> s = hs.signals.Signal1D(range(1000))
>>> s.integrate_in_range() #doctest: +SKIP

Using the CLI

```python
>>> s_int = s.integrate_in_range(signal_range=(560, None))
```

Selecting a range in the axis units, by specifying the signal range with floats.

```python
>>> s_int = s.integrate_in_range(signal_range=(560., 590.))
```

Selecting a range using the index, by specifying the signal range with integers.

```python
>>> s_int = s.integrate_in_range(signal_range=(100, 120))
```

`interpolate_in_between(start, end, delta=3, parallel=None, show_progressbar=None, **kwargs)`

Replace the data in a given range by interpolation. The operation is performed in place.

**Parameters**

- `end (start,)` – 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
- `show_progressbar (None or bool)` – If True, display a progress bar. If None, the default from the preferences settings is used.
- `parallel (None or bool)` – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.
- `extra keyword arguments are passed to (All)`
- `See the function documentation (scipy.interpolate.interp1d).` –
- `details. (for)` –

**Raises** `SignalDimensionError` – If the signal dimension is not 1.

`remove_background(signal_range='interactive', background_type='PowerLaw', polynomial_order=2, fast=True, zero_fill=False, plot_remainder=True, show_progressbar=None, display=True, toolkit=None)`

Remove the background, either in place using a gui or returned as a new spectrum using the command line.

**Parameters**

- `signal_range ("interactive", tuple of ints or floats, 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, the a spectrum will be returned.
- `background_type (str)` – The type of component which should be used to fit the background. Possible components: PowerLaw, Gaussian, Offset, Polynomial 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 possibly 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.

• **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 strings 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.

**Examples**

Using gui, replaces spectrum \(\mathbf{s}\)

```python
>>> s = hs.signals.Signal1D(range(1000))
>>> s.remove_background()  # doctest: +SKIP
```

Using command line, returns a spectrum

```python
>>> s1 = s.remove_background(signal_range=(400,450), background_type='PowerLaw')
```

Using a full model to fit the background

```python
>>> s1 = s.remove_background(signal_range=(400,450), fast=False)
```

Raises *SignalDimensionError* – If the signal dimension is not 1.

**shift1D** (*shift_array, interpolation_method=’linear’, crop=True, expand=False, fill_value=nan, parallel=None, show_progressbar=None*)

Shift the data in place over the signal axis by the amount specified by an array.

**Parameters**

• **shift_array** (*numpy array*) – An array containing the shifting amount. It must have `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.

• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

**Raises** *SignalDimensionError* – If the signal dimension is not 1.

**smooth_lowess** (*smoothing_parameter=None, number_of_iterations=None, show_progressbar=None, parallel=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.

• **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. 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 strings 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.

• *ImportError* – If statsmodels is not installed.

**Notes**

This method uses the lowess algorithm from the *statsmodels* library, which needs to be installed to use this method.

**smooth_savitzky_golay** (*polynomial_order=None, window_length=None, differential_order=0, parallel=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.

• **parallel** *(None or bool)* – If True, perform computation in parallel using multiple cores. 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 strings 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.

**Notes**

More information about the filter in `scipy.signal.savgol_filter`.

**smooth_tv** *(smoothing_parameter=None, show_progressbar=None, parallel=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.

• **parallel** *(None or bool)* – If True, perform computation in parallel using multiple cores. 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 strings 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.

**spikes_removal_tool** *(signal_mask=None, navigation_mask=None, display=True, toolkit=None)*

Graphical interface to remove spikes from EELS spectra.

**Parameters**

• **signal_mask** *(boolean array)* – Restricts the operation to the signal locations not marked as True (masked)

• **navigation_mask** *(boolean array)* – Restricts the operation to the navigation locations not marked as True (masked)

• **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 strings 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.
Find peaks along a 1D line.

Function to locate the positive peaks in a noisy x-y data set. Detects peaks by looking for downward zero-crossings in the first derivative that exceed 'slope_thresh'. Returns an array containing position, height, and width of each peak. Sorted by position. ‘slope_thresh’ and ‘amp_thresh’, control sensitivity: higher values will neglect wider peaks (slope) and smaller features (amp), respectively.

Parameters

- **y** *(array) – 1D input array, e.g. a spectrum*
- **x** *(array (optional)) – 1D array describing the calibration of y (must have same shape as y)*
- **slope_thresh** *(float (optional)) – 1st derivative threshold to count the peak; higher values will neglect broader features; default is set to 0.*
- **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 (optional)) – median filter window to apply to smooth the data (see scipy.signal.medfilt); if 0, no filter will be applied; default is set to 5.*
- **peakgroup** *(int (optional)) – number of points around the “top part” of the peak that are taken to estimate the peak height; for spikes or very narrow peaks, keep PeakGroup=1 or 2; for broad or noisy peaks, make PeakGroup larger to reduce the effect of noise; default is set to 10.*
- **maxpeakn** *(int (optional)) – number of maximum detectable peaks; default is set to 30000.*
- **subchannel** *(bool (optional)) – default is set to True.*


Return type structured array of shape (npeaks)

Examples

```python
>>> x = np.arange(0, 50, 0.01)
>>> y = np.cos(x)
>>> peaks = find_peaks_ohaver(y, x, 0, 0)
```

Notes


hyserspy._signals.signal1d.interpolate1D(number_of_interpolation_points, data)
**hyperspy._signals.signal2d module**

```python
class hyperspy._signals.signal2d.LazySignal2D(*args, **kwargs)
Bases: hyperspy._signals.lazy.LazySignal, hyperspy._signals.signal2d.Signal2D

class hyperspy._signals.signal2d.Signal2D(*args, **kw)
Bases: hyperspy.signal.BaseSignal, hyperspy._signals.common_signal2d.CommonSignal2D

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, roi=None, sobel=True, med_filter=True, hanning=True, plot=False, normalize_corr=False, reference='current', dtype='float', correlation_threshold=None, chunk_size=30, interpolation_order=1, sub_pixel_factor=1, show_progressbar=None, parallel=None)
Align the images in place using user provided shifts or by estimating the shifts.

Please, see `estimate_shift2D` docstring for details on the rest of the parameters not documented in the following section

Parameters

- **crop** (*bool*) – If True, the data will be cropped not to include regions with missing data.
- **fill_value** (*int, float, nan*) – The areas with missing data are filled with the given value. Default is nan.
- **shifts** (*None or list of tuples*) – 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, default 1.*) – The order of the spline interpolation. Default is 1, linear interpolation.
- **parallel** (*None or bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

Returns shifts – The shifts are returned only if `shifts` is None

Return type  np.array

Notes

The statistical analysis approach to the translation estimation when using `reference`='stat' roughly follows [1]. If you use it please cite their article.
create_model (dictionary=None)
Create a model for the current signal

Parameters
dictionary ({None, dict}, optional) – A dictionary to be used to recreate a model. Usually generated using hyperspy.model.as_dictionary()

Returns
Return type: A Model class
crop_image (top=None, bottom=None, left=None, right=None, convert_units=False)
Crops an image in place.

Parameters
- bottom, left, right (top, ) – 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.
- also (See)
- ---------
- 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 bidimensional 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 keyword.

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, ‘auto’, float)) – This parameter is only relevant when reference is ‘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, int)) – If int and ‘reference==’stat’ the number of images used as reference are limited to the given value.
- roi (tuple of ints or floats (left, right, top, bottom)) – Define the region of interest. If int(float) the position is given axis index(value).
- sobel (bool) – apply a sobel filter for edge enhancement
- medfilter (bool) – apply a median filter for noise reduction
- hanning (bool) – Apply a 2d hanning filter
• **plot** *(bool or "reuse")* – 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 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/sub_pixel_factor parts of a pixel. Default is 1, i.e. no sub-pixel accuracy.

**Returns**

**Return type**  list of applied shifts

**Notes**

The statistical analysis approach to the translation estimation when using ‘reference’=’stat’ roughly follows [1]. If you use it please cite their article.

**References**


**plot** *(colorbar=True, scalebar=True, scalebar_color='white', axes_ticks=None, axes_off=False, saturated_pixels=None, vmin=None, vmax=None, gamma=1.0, no_nans=False, 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 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.

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.

If None, no navigator will be provided.

Alternatively a BaseSignal (or subclass) instance can be provided. The signal dimension must be 1 (for a spectrum navigator) or 2 (for a image navigator) and navigation shape must be 0 (for a static navigator) or 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.

colorbar [bool, optional] If true, a colorbar is plotted for non-RGB images.

scalebar [bool, optional] If True and the units and scale of the x and y axes are the same a scale bar is plotted.

colorbar_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.

saturated_pixels: scalar The percentage of pixels that are left out of the bounds. For example, the low and high bounds of a value of 1 are the 0.5% and 99.5% percentiles. It must be in the [0, 100] range. If None (default value), the value from the preferences is used.

vmin, vmax [scalar, optional] vmin and vmax are used to normalize luminance data.

no_nans [bool, optional] If True, set nans to zero for plotting.

centre_colormap [{"auto", True, False}] 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] 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 Only for Signal2D: additional (optional) keyword arguments for matplotlib.pyplot.imshow().

hyperspy._signals.signal2d.estimate_image_shift (ref, image, roi=None, sobel=True, medfilter=True, hanning=True, plot=False, dtype='float', normalize_corr=False, sub_pixel_factor=1, return_maxval=True)

Estimate the shift in a image using phase correlation

This method can only estimate the shift by comparing bidimensional features that should not change the position in the given axis. 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 keyword.

Parameters
HyperSpy Documentation, Release 1.5.1.dev

• **ref** (*2D numpy.ndarray*) – Reference image

• **image** (*2D numpy.ndarray*) – Image to register

• **roi** (*tuple of ints (top, bottom, left, right]*) – Define the region of interest

• **sobel** (*bool*) – apply a sobel filter for edge enhancement

• **medfilter** (*bool*) – apply a median filter for noise reduction

• **hanning** (*bool*) – Apply a 2d hanning filter

• **plot** (*bool | matplotlib.Figure*) – If True, plots the images after applying the filters and the phase correlation. If a figure instance, the images will be plotted to the given figure.

• **reference** (*'current' | 'cascade'*) – If ‘current’ (default) the image at the current coordinates is taken as reference. If ‘cascade’ each image is aligned with the previous one.

• **dtype** (*str or dtype*) – Typecode or data-type in which the calculations must be performed.

• **normalize_corr** (*bool*) – If True use phase correlation instead of standard correlation

• **sub_pixel_factor** (*float*) – Estimate shifts with a sub-pixel accuracy of 1/sub_pixel_factor parts of a pixel. Default is 1, i.e. no sub-pixel accuracy.

**Returns**

• **shifts** (*np.array*) – containing the estimate shifts

• **max_value** (*float*) – The maximum value of the correlation

**Notes**

The statistical analysis approach to the translation estimation when using ‘reference’=’stat’ roughly follows [1]_. If you use it please cite their article.

**References**

hyperspy._signals.signal2d.fft_correlation(*in1, in2, normalize=False*)

Correlation of two N-dimensional arrays using FFT.

Adapted from scipy’s fftconvolve.

**Parameters**

• **in2** (*in1,*) –

• **normalize** (*bool*) – If True performs phase correlation

hyperspy._signals.signal2d.hanning2d(*M, N*)

A 2D hanning window created by outer product.

hyperspy._signals.signal2d.shift_image(*im, shift=0, interpolation_order=1, fill_value=nan*)

hyperspy._signals.signal2d.sobel_filter(*im*)

hyperspy._signals.signal2d.triu_indices_minus_diag(*n*)

Returns the indices for the upper-triangle of an (n, n) array excluding its diagonal

**Parameters**

• **n** (*int*) – The length of the square array
Module contents

hyperspy.datasets package

Submodules

hyperspy.datasets.artificial_data module

Functions for generating artificial data.
For use in things like docstrings or to test HyperSpy functionalities.

**hyperspy.datasets.artificial_data.get_atomic_resolution_tem_signal2d()**
Get an artificial atomic resolution TEM Signal2D.

<table>
<thead>
<tr>
<th>Returns</th>
<th>artificial_tem_image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>HyperSpy Signal2D</td>
</tr>
</tbody>
</table>

**Example**

```python
>>> s = hs.datasets.artificial_data.get_atomic_resolution_tem_signal2d()
>>> s.plot()
```

**hyperspy.datasets.artificial_data.get_core_loss_eels_line_scan_signal()**
Get an artificial core loss electron energy loss line scan spectrum.
Similar to a Mn-L32 and Fe-L32 edge from a perovskite oxide.

<table>
<thead>
<tr>
<th>Returns</th>
<th>artificial_core_loss_line_scan_signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return type</td>
<td>HyperSpy EELSSpectrum</td>
</tr>
</tbody>
</table>

**Example**

```python
>>> s = hs.datasets.artificial_data.get_core_loss_eels_line_scan_signal()
>>> s.plot()
```

See also:

- **get_low_loss_eels_model()** get a low loss signal
- **get_core_loss_eels_model()** get a model instead of a signal
- **get_low_loss_eels_line_scan_signal()** get low loss signal with the same size

**hyperspy.datasets.artificial_data.get_core_loss_eels_model**(add_powerlaw=False)
Get an artificial core loss electron energy loss model.
Similar to a Mn-L32 edge from a perovskite oxide.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>add_powerlaw (bool) – If True, adds a powerlaw background to the spectrum. Default False.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>artificial_core_loss_model</td>
</tr>
<tr>
<td>Return type</td>
<td>HyperSpy EELSMoodel</td>
</tr>
</tbody>
</table>
Example

```python
>>> import hs.datasets.artificial_data as ad
>>> s = ad.get_core_loss_eels_model()
>>> s.plot()
```

With the powerlaw background

```python
>>> s = ad.get_core_loss_eels_model(add_powerlaw=True)
>>> s.plot()
```

See also:

- `get_low_loss_eels_model()` - get a low loss signal
- `get_core_loss_eels_signal()` - get a model instead of a signal

```python
hyperspy.datasets.artificial_data.get_core_loss_eels_signal(add_powerlaw=False)
```

Get an artificial core loss electron energy loss spectrum.

Similar to a Mn-L32 edge from a perovskite oxide.

Some random noise is also added to the spectrum, to simulate experimental noise.

**Parameters**

- `add_powerlaw (bool)` – If True, adds a powerlaw background to the spectrum. Default False.

**Returns**

- `artificial_core_loss_signal`

**Return type**

HyperSpy EELSSpectrum

Example

```python
>>> import hs.datasets.artificial_data as ad
>>> s = ad.get_core_loss_eels_signal()
>>> s.plot()
```

With the powerlaw background

```python
>>> s = ad.get_core_loss_eels_signal(add_powerlaw=True)
>>> s.plot()
```

To make the noise the same for multiple spectra, which can be useful for testing fitting routines

```python
>>> np.random.seed(seed=10)
>>> s1 = ad.get_core_loss_eels_signal()
>>> np.random.seed(seed=10)
>>> s2 = ad.get_core_loss_eels_signal()
>>> (s1.data == s2.data).all()
```

True

See also:

- `get_low_loss_eels_model()` - get a low loss signal
- `get_core_loss_eels_model()` - get a model instead of a signal
- `get_low_loss_eels_line_scan_signal()` - get EELS low loss line scan
**get_low_loss_eels_line_scan_signal()**  get EELS low loss line scan

ehyperspy.datasets.artificial_data.get_low_loss_eels_line_scan_signal()

Get an artificial low loss electron energy loss line scan spectrum.

The zero loss peak is offset by 4.1 eV.

Returns artificial_low_loss_line_scan_signal

Return type HyperSpy EELSSpectrum

Example

```python
>>> s = hs.datasets.artificial_data.get_low_loss_eels_signal()
>>> s.plot()
```

See also:

**get_core_loss_eels_signal()**  get a core loss signal
**get_core_loss_eels_model()**  get a core loss model
**get_core_loss_eels_line_scan_signal()**  core loss signal with the same size

hyperspy.datasets.artificial_data.get_low_loss_eels_signal()

Get an artificial low loss electron energy loss spectrum.

The zero loss peak is offset by 4.1 eV.

Returns artificial_low_loss_signal

Return type HyperSpy EELSSpectrum

Example

```python
>>> s = hs.datasets.artificial_data.get_low_loss_eels_signal()
>>> s.plot()
```

See also:

**get_core_loss_eels_signal()**  get a core loss signal
**get_core_loss_eels_model()**  get a core loss model
**get_low_loss_eels_line_scan_signal()**  get EELS low loss line scan
**get_core_loss_eels_line_scan_signal()**  get EELS core loss line scan

hyperspy.datasets.example_signals module

Module contents

The **hyperspy.datasets** module includes access to local and remote datasets.

Functions:

- **eelsdb**  Download spectra from the EELS data base [http://eelsdb.eu](http://eelsdb.eu)
Submodules:
The `datasets` module contains the following submodules:

- **example_signals**  Example datasets distributed with HyperSpy.

**hyperspy.docstrings package**

**Submodules**

- **hyperspy.docstrings.parameters module**

  Common docstring snippets for parameters.

- **hyperspy.docstrings.plot module**

  Common docstring snippets for plot.

- **hyperspy.docstrings.signal module**

  Common docstring snippets for signal.

- **hyperspy.docstrings.signal1d module**

  Common docstring snippets for signal1d.

**Module contents**

Common docstring snippets.

**hyperspy.drawing package**

**Subpackages**

- **hyperspy.drawing._markers package**

**Submodules**

- **hyperspy.drawing._markers.horizontal_line module**

  **class** `hyperspy.drawing._markers.horizontal_line.HorizontalLine(y, **kwargs)`

  Bases: `hyperspy.drawing.marker.MarkerBase`

  Horizontal line marker that can be added to the signal figure

  **Parameters**
• **y (array or float)** – The position of the line. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.

• **kwargs** – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> s = hs.signals.Signal1D(np.random.random([10, 100])) * 10
>>> m = hs.plot.markers.horizontal_line(y=range(10), color='green')
>>> s.add_marker(m)
```

Adding a marker permanently to a signal

```python
>>> s = hs.signals.Signal1D(np.random.random([10, 100]))
>>> m = hs.plot.markers.horizontal_line(y=5, color='green')
>>> s.add_marker(m, permanent=True)
```

**hyperspy.drawing._markers.horizontal_line_segment module**

```python
hyperspy.drawing._markers.horizontal_line_segment.HorizontalLineSegment(x1, x2, y, **kwargs)
```

**Bases:** hyperspy.drawing.marker.MarkerBase

Horizontal line segment marker that can be added to the signal figure

**Parameters**

• **x1 (array or float)** – The position of the start of the line segment in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.

• **x2 (array or float)** – The position of the end of the line segment in x. see x1 arguments

• **y (array or float)** – The position of line segment in y. see x1 arguments

• **kwargs** – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> im = hs.signals.Signal2D(np.zeros((100, 100)))
>>> m = hs.plot.markers.horizontal_line_segment(x1=20, x2=70, y=70, linewidth=4, color='red', linestyle='dotted')
>>> im.add_marker(m)
```

Adding a marker permanently to a signal

```python
>>> im = hs.signals.Signal2D(np.zeros((100, 100)))
>>> m = hs.plot.markers.horizontal_line_segment(x1=10, x2=30, y=42, linewidth=4, color='red', linestyle='dotted')
>>> im.add_marker(m, permanent=True)
```
**update()**

**hyperspy.drawing._markers.line_segment module**

```python
class hyperspy.drawing._markers.line_segment.LineSegment(x1, y1, x2, y2, **kwargs)
```

Bases: `hyperspy.drawing.marker.MarkerBase`

Line segment marker that can be added to the signal figure

**Parameters**

- `x1 (array or float)` – The position of the start of the line segment in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.
- `y1 (array or float)` – The position of the start of the line segment in y. see x1 arguments
- `x2 (array or float)` – The position of the end of the line segment in x. see x1 arguments
- `y2 (array or float)` – The position of the end of the line segment in y. see x1 arguments
- `kwargs` – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> im = hs.signals.Signal2D(np.zeros((100, 100)))
>>> m = hs.plot.markers.line_segment(
    x1=20, x2=70, y1=20, y2=70,
    linewidth=4, color='red', linestyle='dotted')
>>> im.add_marker(m)
```

Permanently adding a marker to a signal

```python
>>> im = hs.signals.Signal2D(np.zeros((100, 100)))
>>> m = hs.plot.markers.line_segment(
    x1=10, x2=30, y1=50, y2=70,
    linewidth=4, color='red', linestyle='dotted')
>>> im.add_marker(m, permanent=True)
```

**hyperspy.drawing._markers.point module**

```python
class hyperspy.drawing._markers.point.Point(x, y, size=20, **kwargs)
```

Bases: `hyperspy.drawing.marker.MarkerBase`

Point marker that can be added to the signal figure.

If the signal has one or several navigation axes, the point marker can change as a function of the navigation position. This done by using an array for the x and y parameters. This array must have the same shape as the navigation axes of the signal.

**Parameters**
• **x** (*array or float*) – The position of the point in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.

• **y** (*array or float*) – The position of the point in y. see x arguments

• **size** (*array or float, optional, default 20*) – The size of the point. see x arguments

• **kwargs** – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> im = hs.signals.Signal2D(np.random.random([10, 50, 50]))
>>> m = hs.plot.markers.point(x=range(10), y=range(10)[::-1],
                             color='red')
>>> im.add_marker(m)
```

Adding a marker permanently to a signal

```python
>>> im = hs.signals.Signal2D(np.random.random([10, 50, 50]))
>>> m = hs.plot.markers.point(10, 30, color='blue', size=50)
>>> im.add_marker(m, permanent=True)
```

Markers on local maxima

```python
>>> from skimage.feature import peak_local_max
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.ascent()).as_signal2D([2,0])
>>> index = array([peak_local_max(i.data, min_distance=100, num_peaks=4)
                  for i in im])
>>> for i in range(4):
>>>     m = hs.plot.markers.point(x=index[:, i, 1],
                                y=index[:, i, 0], color='red')
>>> im.add_marker(m)
```

**update()**

**hyperspy.drawing._markers.rectangle module**

```python
class hyperspy.drawing._markers.rectangle.Rectangle(xl, yl, x2, y2, **kwargs)
Bases: hyperspy.drawing.marker.MarkerBase
```

Rectangle marker that can be added to the signal figure

**Parameters**

• **xl** (*array or float*) – The position of the up left corner of the rectangle in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.

• **yl** (*array or float*) – The position of the up left corner of the rectangle in y. see xl arguments

• **x2** (*array or float*) – The position of the down right corner of the rectangle in x. see xl arguments

• **y2** (*array or float*) – The position of the down right corner of the rectangle in y. see yl arguments
• \texttt{y2 (array or float)} – The position of the down right of the rectangle in y. see \texttt{x1} arguments

• \texttt{kwarg}t – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> m = hs.plot.markers.rectangle(x1=150, y1=100, x2=400, y2=400,
                                         color='red')
>>> im.add_marker(m)
```

Adding a marker permanently to a signal

```python
>>> im = hs.signals.Signal2D(np.random.random((50, 50))
>>> m = hs.plot.markers.rectangle(x1=20, y1=30, x2=40, y2=49)
>>> im.add_marker(m, permanent=True)
```

**hyperspy.drawing._markers.text module**

**class hyperspy.drawing._markers.text.Text (x, y, text, **kwargs)**

Bases: hyperspy.drawing.marker.MarkerBase

Text marker that can be added to the signal figure

**Parameters**

• \texttt{x (array or float)} – The position of the text in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.

• \texttt{y (array or float)} – The position of the text in y. see \texttt{x} arguments

• \texttt{text (array or str)} – The text. see \texttt{x} arguments

• \texttt{kwarg}t – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> s = hs.signals.Signal1D(np.arange(100).reshape([10, 10]))
>>> s.plot(navigator='spectrum')
>>> for i in range(10):
>>>     m = hs.plot.markers.text(y=range(50, 1000, 100)[i],
                                 x=i, text='abcdefghij'[i])
>>>     s.add_marker(m, plot_on_signal=False)
>>> m = hs.plot.markers.text(x=5, y=range(7, 110, 10),
                           text=[i for i in 'abcdefghij'])
>>> s.add_marker(m)
```

Add a marker permanently to a signal
```python
>>> s = hs.signals.Signal1D(np.arange(100).reshape([10, 10]))
>>> m = hs.plot.markers.text(5, 5, "a_text")
>>> s.add_marker(m, permanent=True)
```

**update()**

### hyperspy.drawing._markers.vertical_line module

class hyperspy.drawing._markers.vertical_line.VerticalLine(x, **kwargs)

Bases: hyperspy.drawing.marker.MarkerBase

Vertical line marker that can be added to the signal figure

Parameters

- **x (array or float)** – The position of the line. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.
- **kwargs** – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> s = hs.signals.Signal1D(np.random.random([10, 100]))
>>> m = hs.plot.markers.vertical_line(x=range(10), color='green')
>>> s.add_marker(m)
```

Adding a marker permanently to a signal

```python
>>> s = hs.signals.Signal1D(np.random.random([100, 100]))
>>> m = hs.plot.markers.vertical_line(x=30)
>>> s.add_marker(m, permanent=True)
```

**update()**

### hyperspy.drawing._markers.vertical_line_segment module

class hyperspy.drawing._markers.vertical_line_segment.VerticalLineSegment(x, y1, y2, **kwargs)

Bases: hyperspy.drawing.marker.MarkerBase

Vertical line segment marker that can be added to the signal figure

Parameters

- **x (array or float)** – The position of line segment in x. If float, the marker is fixed. If array, the marker will be updated when navigating. The array should have the same dimensions in the navigation axes.
- **y1 (array or float)** – The position of the start of the line segment in x. see x1 arguments
- **y2 (array or float)** – The position of the start of the line segment in y. see x1 arguments
• **kwargs – Keywords argument of axvline valid properties (i.e. recognized by mpl.plot).

**Example**

```python
>>> im = hs.signals.Signal2D(np.zeros((100, 100)))
>>> m = hs.plot.markers.vertical_line_segment(x=20, y1=30, y2=70, linewidth=4, color='red', linestyle='dotted')
>>> im.add_marker(m)
```

Add a marker permanently to a marker

```python
>>> im = hs.signals.Signal2D(np.zeros((60, 60)))
>>> m = hs.plot.markers.vertical_line_segment(x=10, y1=20, y2=50)
>>> im.add_marker(m, permanent=True)
```

**Module contents**

**hyperspy.drawing._widgets package**

**Submodules**

**hyperspy.drawing._widgets.circle module**

```python
class hyperspy.drawing._widgets.circle.CircleWidget(axes_manager, **kwargs)
```

CircleWidget is a symmetric, Cicle-patch based widget, which can be dragged, and resized by keystrokes/code.

```python
def decrease_size()
    Decrement all sizes by one step. Applied via ‘size’ property.

def get_centre()
    Get’s the center indices. The default implementation is simply the position + half the size in axes space, which should work for any symmetric widget, but more advanced widgets will need to decide whether to return the center of gravity or the geometrical center of the bounds.

def get_size_in_indices()
    Gets the size property converted to the index space (via ‘axes’ attribute).

def increase_size()
    Increment all sizes by one step. Applied via ‘size’ property.
```

**hyperspy.drawing._widgets.horizontal_line module**

```python
class hyperspy.drawing._widgets.horizontal_line.HorizontalLineWidget(axes_manager, **kwargs)
Bases: hyperspy.drawing.widget.Widget1DBase
```

A draggable, horizontal line widget.
hyperspy.drawing._widgets.label module

class hyperspy.drawing._widgets.label.LabelWidget(axes_manager, color='black', **kwargs)

Bases: hyperspy.drawing.widget.Widget1DBase

A draggable text widget. Adds the attributes ‘string’ and ‘bbox’. These are all arguments for matplotlib’s Text artist. The default y-coordinate of the label is set to 0.9.

property string

hyperspy.drawing._widgets.line2d module

class hyperspy.drawing._widgets.line2d.Line2DWidget(axes_manager, **kwargs)

Bases: hyperspy.drawing.widget.ResizableDraggableWidgetBase

A free-form line on a 2D plot. Enables dragging and moving the end points, but also allows rotation of the widget by moving the mouse beyond the end points of the line.

The widget adds the ‘linewidth’ attribute, which is different from the size in the following regards: ‘linewidth’ is simply the width of the patch drawn from point to point. If ‘size’ is greater than 1, it will in principle select a rotated rectangle. If ‘size’ is greater than 4, the bounds of this rectangle will be visualized by two dashed lines along the outline of this rectangle, instead of a single line in the center.

The widget also adds the attributes ‘radius_resize’, ‘radius_move’ and ‘radius_rotate’ (defaults: 5, 5, 10), which determines the picker radius for resizing, aka. moving the edge points (by picking within ‘radius_resize’ from an edge point); for moving (by picking within ‘radius_move’ from the body of the line); and for rotation (by picking within ‘radius_rotate’ of the edge points on the “outside” of the line). The priority is in the order resize, rotate, move; so the ‘radius_rotate’ should always be larger than ‘radius_resize’ if the function is to be accessible (putting it lower is an easy way to disable the functionality).

NOTE: This widget’s internal position does not lock to axes points by default.

NOTE: The ‘position’ is now a 2D tuple: tuple(tuple(x1, x2), tuple(y1, y2)) NOTE: The ‘size’ property corresponds to line width, so it has a len() of only one.

FUNC_A = 32
FUNC_B = 64
FUNC_MOVE = 1
FUNC_NONE = 0
FUNC_RESIZE = 2
FUNC_ROTATE = 4
FUNC_SIZERS = 8

connect_navigate()
    Connect to the axes_manager such that changes in the widget or in the axes_manager are reflected in the other.

get_centre()
    Get the line center, which is simply the mean position of its vertices.

get_line_length()
    Returns line length in axes coordinates. Requires units on all axes to be the same to make any physical sense.
**onpick** *(event)*

Pick, and if picked, figure out which function to apply. Also store pouse position for use by _onmouse-move. As rotation does not work very well with incremental rotations, the original points are stored if we’re rotating.

```
hyperspy.drawing._widgets.line2d.angle_between(v1, v2)
```

Returns the angle in radians between @D vectors ‘v1’ and ‘v2’:

```
>>> angle_between((1, 0), (0, 1))
1.5707963267948966
>>> angle_between((1, 0), (1, 0))
0.0
>>> angle_between((1, 0), (-1, 0))
3.141592653589793
```

```
hyperspy.drawing._widgets.line2d.unit_vector(vector)
```

Returns the unit vector of the vector.

**hyperspy.drawing._widgets.range module**

```
class hyperspy.drawing._widgets.range.ModifiableSpanSelector(ax, **kwargs)
    Bases: matplotlib.widgets.SpanSelector
    contains (mouseevent)
    draw_patch (*args)
        Update the patch drawing.
    dummy (*args, **kwargs)
    mm_on_press (event)
    mm_on_release (event)
    move_left (event)
    move_rect (event)
    move_right (event)
    property range
    release (event)
        When the button is released, the span stays in the screen and the iteractivity machinery passes to modify mode
    set_initial (initial_range=None)
        Remove selection events, set the spanner, and go to modify mode.
    switch_left_right (x, left_to_right)
    turn_off ()
    update (*args)
        draw using newfangled blit or oldfangled draw depending on useblit
    update_range ()
```

```
class hyperspy.drawing._widgets.range.RangeWidget(axes_manager, ax=None, alpha=0.5, **kwargs)
    Bases: hyperspy.drawing.widget.ResizableDraggableWidgetBase

    RangeWidget is a span-patch based widget, which can be dragged and resized by mouse/keys. Basically a wrapper for ModifiablepanSelector so that it conforms to the common widget interface.
```
For optimized changes of geometry, the class implements two methods ‘set_bounds’ and ‘set_ibounds’, to set the geometry of the rectangle by value and index space coordinates, respectively. Implements the internal method _validate_geometry to make sure the patch will always stay within bounds.

**disconnect**(*)
Disconnect from all events (both matplotlib and navigation).

**set_bounds**(*args, **kwargs*)
Set bounds by values. Bounds can either be specified in order left, bottom, width, height; or by keywords:

- ‘bounds’: tuple (left, width)
- OR ‘x’/’left’ * ‘w’/’width’, alternatively ‘right’ (x+w)

If specifying with keywords, any unspecified dimensions will be kept constant (note: width will be kept, not right).

**set_ibounds**(*args, **kwargs*)
Set bounds by indices. Bounds can either be specified in order left, bottom, width, height; or by keywords:

- ‘bounds’: tuple (left, width)
- OR ‘x’/’left’ * ‘w’/’width’, alternatively ‘right’

If specifying with keywords, any unspecified dimensions will be kept constant (note: width will be kept, not right).

**set_on**(value)
Change the on state of the widget. If turning off, all patches will be removed from the matplotlib axes and the widget will disconnect from all events. If turning on, the patch(es) will be added to the matplotlib axes, and the widget will connect to its default events.

**hyperspy.drawing._widgets.range.in_interval**(number, interval)

**hyperspy.drawing._widgets.rectangles module**

**class** hyperspy.drawing._widgets.rectangles.RectangleWidget(axes_manager, **kwargs)

_Bases_: hyperspy.drawing._widgets.rectangles.SquareWidget, hyperspy.drawing.widget.ResizersMixin

RectangleWidget is an asymmetric, Rectangle-patch based widget, which can be dragged and resized by mouse/keys. For resizing by mouse, it adds a small Rectangle patch on the outer border of the main patch, to serve as resize handles. This feature can be enabled/disabled by the ‘resizers’ property, and the size/color of the handles are set by ‘resize_color’/’resize_pixel_size’.

For optimized changes of geometry, the class implements two methods ‘set_bounds’ and ‘set_ibounds’, to set the geometry of the rectangle by value and index space coordinates, respectively. It also adds the ‘width’ and ‘height’ properties for verbosity.

For keyboard resizing, ‘x’/’c’ and ‘y’/’u’ will increase/decrease the size of the rectangle along the first and the second axis, respectively.

Implements the internal method _validate_geometry to make sure the patch will always stay within bounds.

**property height**

**on_key_press**(event)

**set_bounds**(*args, **kwargs*)
Set bounds by values. Bounds can either be specified in order left, bottom, width, height; or by keywords:
HyperSpy Documentation, Release 1.5.1.dev

- 'bounds': tuple (left, top, width, height)

  OR * 'x'/left * 'y'/top * 'w'/width', alternatively 'right' (x+w) * 'h'/height', alternatively 'bottom' (y+h)

  If specifying with keywords, any unspecified dimensions will be kept constant (note: width/height will be kept, not right/bottom).

**set_ibounds (**args, **kwargs)**

Set bounds by indices. Bounds can either be specified in order left, bottom, width, height; or by keywords:

- 'bounds': tuple (left, top, width, height)

  OR * 'x'/left * 'y'/top * 'w'/width', alternatively 'right' * 'h'/height', alternatively 'bottom'

  If specifying with keywords, any unspecified dimensions will be kept constant (note: width/height will be kept, not right/bottom).

**property width**

class hyperspy.drawing._widgets.rectangles.SquareWidget (axes_manager, **kwargs)

Bases: hyperspy.drawing.widget.Widget2DBase

SquareWidget is a symmetric, Rectangle-patch based widget, which can be dragged, and resized by keystrokes/code. As the widget is normally only meant to indicate position, the sizing is deemed purely visual, but there is nothing that forces this use. However, it should be noted that the outer bounds only correspond to pure indices for odd sizes.

**hyperspy.drawing._widgets.scalebar module**

class hyperspy.drawing._widgets.scalebar.ScaleBar (ax, units, pixel_size=None, color='white', position=None, max_size_ratio=0.25, lw=2, length=None, animated=False)

Bases: object

Add a scale bar to an image.

**ax** [matplotlib axes] The axes where to draw the scale bar.

**units** : string **pixel_size** : {None, float}

  If None the axes of the image are supposed to be calibrated. Otherwise the pixel size must be specified.

**color** : a valid matplotlib color position {None, (float, float)}

  If None the position is automatically determined.

**max_size_ratio** [float] The maximum size of the scale bar in respect to the length of the x axis

**lw** [int] The line width

**length** [{None, float}] If None the length is automatically calculated using the max_size_ratio.

**calculate_line_position** (pad=0.05)

**calculate_size** (max_size_ratio=0.25)

**calculate_text_position** (pad=0.01)

**get_units_string**()

**plot_scale** (line_width=1)
remove()
set_color(c)
set_length(length)
set_tex_bold()

**hyperspy.drawing._widgets.vertical_line** module

class hyperspy.drawing._widgets.vertical_line.VerticalLineWidget(axes_manager, **kwargs)

Bases: hyperspy.drawing.widget.Widget1DBase

A draggable, vertical line widget.

**Module contents**

**Submodules**

**hyperspy.drawing.figure** module

class hyperspy.drawing.figure.BlittedFigure

Bases: object

add_marker(marker)
close()

create_figure(**kwargs)

Create matplotlib figure

Parameters **kwargs – All keyword arguments are passed to plt.figure.

property title

**hyperspy.drawing.image** module

class hyperspy.drawing.image.ImagePlot

Bases: hyperspy.drawing.figure.BlittedFigure

property axes_ticks

configure()

connect()

create_axis()

create_figure(max_size=None, min_size=2, **kwargs)

Create matplotlib figure

The figure size is automatically computed by default, taking into account the x and y dimensions of the image. Alternatively the figure size can be defined by passing the figsize keyword argument.

Parameters

* min_size (max_size,) – The maximum and minimum size of the axes in inches.

These have no effect when passing the figsize keyword to manually set the figure size.
• **kwargs – All keyword arguments are passed to `plt.figure`.

disconnect()

gui_adjust_contrast (`display=True, toolkit=None`)  
Display widgets to adjust image contrast if available.  
:param display: If True, display the user interface  
widgets. If False, return the  
widgets container in a dictionary, usually for customisation or testing.

Parameters toolkit (`str, iterable of strings 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.

on_key_press (`event`)  

optimize_colorbar (`number_of_ticks=5, tolerance=5, step_prec_max=1`)  

optimize_contrast (`data, ignore_user_values=False`)  

plot (`data_function_kwargs={}, **kwargs`)  

property scalebar  

set_contrast (`vmin, vmax`)  

set_quantity_label()  

toggle_norm()  

update (`data_changed=True, **kwargs`)  

property vmax  

property vmin

**hyperspy.drawing.marker module**

class hyperspy.drawing.marker.MarkerBase  
Bases: `object`  

Marker that can be added to the signal figure  

marker_properties  
Accepts a dictionary of valid (i.e. recognized by mpl.plot) containing valid line properties. In addition it  
understands the keyword `type` that can take the following values: `{'line', 'text'}`

    Type dictionary  

add_data (**kwargs)  
Add data to the structured array. Each field of data should have the same dimensions than the navigation  
axes. The other fields are not changed.

close (`render_figure=True`)  
Remove and disconnect the marker.

Parameters render_figure (`bool, optional, default True`) – If True, the fig- 
ure 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.

get_data_position (`ind`)
isiterable(obj)

property marker_properties

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.

set_data(x1=None, y1=None, x2=None, y2=None, text=None, size=None)
Set data to the structured array. Each field of data should have the same dimensions than the navigation axes. The other fields are overwritten.

set_marker_properties(**kwargs)
Set the line_properties attribute using keyword arguments.

hyperspy.drawing.marker.dict2marker(marker_dict, marker_name)

hyperspy.drawing.marker.markers_metadata_dict_to_markers(metadata_markers_dict, axes_manager)

hyperspy.drawing.mpl_he module

class hyperspy.drawing.mpl_he.MPL_HyperExplorer
Bases: object

assign_pointer()
close()
close_navigator_plot()
property is_active
plot(**kwargs)
plot_navigator(colorbar=True, scalebar=True, scalebar_color='white', axes_ticks=None, saturated_pixels=None, vmin=None, vmax=None, no_nans=False, centre_colormap='auto', title=None, min_aspect=0.1, **kwds)

plot_signal()

hyperspy.drawing.mpl_hie module

class hyperspy.drawing.mpl_hie.MPL_HyperImage_Explorer
Bases: hyperspy.drawing.mpl_he.MPL_HyperExplorer

plot_signal(colorbar=True, scalebar=True, scalebar_color='white', axes_ticks=None, axes_off=False, saturated_pixels=None, vmin=None, vmax=None, no_nans=False, centre_colormap='auto', norm='auto', min_aspect=0.1, gamma=1.0, linthresh=0.01, linscale=0.1, **kwargs)
Plot image.

Parameters

* colorbar (bool, optional) – If true, a colorbar is plotted for non-RGB images.
- **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.

- **saturated_pixels** *(scalar)* – The percentage of pixels that are left out of the bounds. For example, the low and high bounds of a value of 1 are the 0.5% and 99.5% percentiles. It must be in the [0, 100] range. If None (default value), the value from the preferences is used.

- **vmax**(vmin,) – vmin and vmax are used to normalize luminance data.

- **no_nans** *(bool, optional)* – If True, set nans to zero for plotting.

- **centre_colormap** *("auto", True, False)* – 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)* – 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** – Only for Signal2D: additional (optional) keyword arguments for matplotlib.pyplot.imshow().

**hyperspy.drawing.mpl_hse module**

```python
class hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer
    Bases: hyperspy.drawing.mpl_he.MPL_HyperExplorer

Plots the current spectrum to the screen and a map with a cursor to explore the SI.

    add_right_pointer(**kwargs)
    property auto_update_plot
    key2switch_right_pointer(event)
    plot_signal(**kwargs)
    remove_right_pointer()
    property right_pointer_on
        I’m the ‘x’ property.
```

**hyperspy.drawing.signal module**

**hyperspy.drawing.signal1d module**

```python
class hyperspy.drawing.signal1d.Signal1DFigure(title="")
    Bases: hyperspy.drawing.figure.BlittedFigure

    add_line(line, ax='left')
    create_axis()
    create_right_axis()
```
plot (data_function_kwargs={}, **kwargs)

update ()

class hyperspy.drawing.signal1d.Signal1DLine

Bases: object

Line that can be added to Signal1DFigure.

type

Select the line drawing style.

Type {'scatter', 'step', 'line'}

line_properties

Accepts a dictionary of valid (i.e. recognized by mpl.plot) containing valid line properties. In addition it understands the keyword type that can take the following values: {'scatter', 'step', 'line'}

Type dictionary

auto_update

If False, executing _auto_update_line does not update the line plot.

Type bool

set_line_properties ()

Enables setting the line_properties attribute using keyword arguments.

Raises ValueError – If an invalid keyword value is passed to line_properties.

close ()

property color

property get_complex

property line_properties

plot (data=1, data_function_kwargs={}, norm='linear')

set_line_properties (**kwargs)

property type

update (force_replot=False, render_figure=True)

Update the current spectrum figure

hyperspy.drawing.tiles module

class hyperspy.drawing.tiles.HistogramTilePlot

Bases: hyperspy.drawing.figure.BlittedFigure

close ()

create_axis (ncols=1, nrows=1, number=1, title="")

plot (db, **kwargs)

update (db, **kwargs)
class hyperspy.drawing.utils.ColorCycle

Bases: object

hyperspy.drawing.utils.animate_legend(fig=None, ax=None)

Animate the legend of a figure.

A spectrum can be toggle on and off by clicking on the legended line.

Parameters

- **fig** (None | matplotlib.figure) – If None pick the current figure using “plt.gcf”
- **ax** (None | matplotlib.axes) – If None pick the current axes using “plt.gca”.

Note: Code inspired from legend_picking.py in the matplotlib gallery

hyperspy.drawing.utils.centre_colormap_values(vmin, vmax)

Calculate vmin and vmax to set the colormap midpoint to zero.

Parameters

- **vmax** (vmin,) – The range of data to display.

Returns cvmin, cvmax – The values to obtain a centre colormap.

Return type scalar

hyperspy.drawing.utils.contrast_stretching(data, saturated_pixels)

Calculate bounds that leaves out a given percentage of the data.

Parameters

- **data** (numpy array) –
- **saturated_pixels** (scalar, None) – The percentage of pixels that are left out of the bounds. For example, the low and high bounds of a value of 1 are the 0.5% and 99.5% percentiles. It must be in the [0, 100] range. If None, set the value to 0.

Returns vmin, vmax – The low and high bounds

Return type scalar

Raises ValueError if the value of saturated_pixels is out of the valid range.

hyperspy.drawing.utils.create_figure(window_title=None, _on_figure_window_close=None, disable_xyscale_keys=False, **kwargs)

Create a matplotlib figure.

This function adds the possibility to execute another function when the figure is closed and to easily set the window title. Any keyword argument is passed to the plt.figure function.

Parameters

- **window_title** (string) –
- **_on_figure_window_close** (function) –
- **disable_xyscale_keys** (bool, disable the k, l and L shortcuts which) –
- **the x or y axis between linear and log scale. (toggle)** –

Returns fig

Return type plt.figure
hyperspy.drawing.utils.key_press_handler_custom(event, canvas)

hyperspy.drawing.utils.make_cmap(colors, name='my_colormap', position=None, bit=False, register=True)

Create a matplotlib colormap with customized colors, optionally registering it with matplotlib for simplified use.

Adapted from Chris Slocum’s code at: https://github.com/CSlocumWX/custom_colormap/blob/master/custom_colormaps.py and used under the terms of that code’s BSD-3 license

Parameters

- **colors** (iterable) – list of either tuples containing rgb values, or html strings. Colors should be arranged so that the first color is the lowest value for the colorbar and the last is the highest.
- **name** (str) – name of colormap to use when registering with matplotlib
- **position** (None or iterable) – list containing the values (from [0,1]) that dictate the position of each color within the colormap. If None (default), the colors will be equally-spaced within the colorbar.
- **bit** (boolean) – True if RGB colors are given in 8-bit [0 to 255] or False if given in arithmetic basis [0 to 1] (default)
- **register** (boolean) – switch to control whether or not to register the custom colormap with matplotlib in order to enable use by just the name string

hyperspy.drawing.utils.on_figure_window_close(figure, function)

Connects a close figure signal to a given function.

Parameters

- **figure** (mpl figure instance)
- **function** (function)

hyperspy.drawing.utils.plot_RGB_map(im_list, normalization='single', dont_plot=False)

Plot 2 or 3 maps in RGB.

Parameters

- **im_list** (list of Signal2D instances)
- **normalization** (‘single’, ‘global’)
- **dont_plot** (bool)

Returns array

Return type RGB matrix

hyperspy.drawing.utils.plot_histograms(signal_list, bins='freedman', range_bins=None, color=None, line_style=None, legend='auto', fig=None, **kwargs)

Plot the histogram of every signal in the list in the same figure.

This function creates a histogram for each signal and plot the list with the `utils.plot.plot_spectra` function.

Parameters

- **signal_list** (iterable) – Ordered spectra list to plot. If style is “cascade” or “mosaic” the spectra can have different size and axes.
- **bins** (int or list or str, optional) – If bins is a string, then it must be one of: ‘knuth’: use Knuth’s rule to determine bins ‘scotts’: use Scott’s rule to determine
bins ‘freedman’ : use the Freedman-diaconis rule to determine bins ‘blocks’ : use bayesian blocks for dynamic bin widths

• **range_bins**(tuple or None, optional.) – the minimum and maximum range for the histogram. If not specified, it will be (x.min(), x.max())

• **color** (valid matplotlib color or a list of them or None, optional.) – Sets the color of the lines of the plots. If 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.

• **line_style** (valid matplotlib line style or a list of them or None, optional.) –

• **optional.** – The main line style are ‘-’,'–','steps','-.',':'. If a list, if its length is less than the number of spectra to plot, line_style will be cycled. If None, use continuous lines, eg: (‘-’,’–’,’steps’,’-.’,’:’)

• **legend**(None or list of str or ‘auto’, optional.) – Display a legend. If ‘auto’, the title of each spectra (metadata.General.title) is used.

• **legend_picking**(bool, optional.) – If true, a spectrum can be toggle on and off by clicking on the legended line.

• **fig**(matplotlib figure or None, optional.) – If None, a default figure will be created.

• **kwargs** – other keyword arguments (weight and density) are described in np.histogram().

**Example**

Histograms of two random chi-square distributions

```python
>>> 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'])
```

Returns **ax** – An array is returned when style is “mosaic”.

Return type matplotlib axes or list of matplotlib axes

hyperspy.drawing.utils.plot_images(images, cmap=None, no_nans=False, per_row=3, label='auto', labelwrap=30, supitle=None, supitle_fontsize=18, colorbar='multi', centre_colormap='auto', saturated_pixels=None, 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, *args, **kwargs)

Plot multiple images as sub-images in one figure.

Extra keyword arguments are passed to matplotlib.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 tranposed to form a Signal2D. Multi-dimensional images will have each plane plotted as a separate image. If any signal shape is not suitable, a ValueError will be raised.
• `cmap` (matplotlib colormap, list, or 'mpl_colors', optional) – The colormap used for the images, by default read from pyplot. 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, or 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 each figure

• `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` ("'multi', None, 'single'”)– Controls the type of colorbars that are plotted. If None, no colorbar is plotted. If ‘multi’ (default), 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

• `centre_colormap` ("auto", True, False) – 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.

• `saturated_pixels` (None, scalar or list of scalar, optional, default: 0) – If list of scalar, the length should match the number of images to show. If provide in the list, set the value to 0. The percentage of pixels that are left out of the bounds. For example, the low and high bounds of a value of 1 are the 0.5% and 99.5% percentiles. It must be in the [0, 100] range.

• `scalebar` ("None", 'all', list of ints), 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 or 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 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 or numeric, 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** *(mpl figure, optional)* – If set, the images will be plotted to an existing MPL figure

- **vmax** *(vmin,)* – If list of scalar, the length should match the number of images to show. A list of scalar is not compatible with a single colorbar. See vmin, vmax of `matplotlib.imshow()` for more details.

**kwargs, optional *(**args,**)* – Additional arguments passed to `matplotlib.imshow()`

Returns **axes_list** – a list of subplot axes that hold the images

Return type **list**

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`

```
hyperspy.drawing.utils.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 instances)* – If sync is set to True, the signals must have the same navigation shape, but not necessarily the same signal shape.

- **sync** *(True or False, default "True")* – If True: 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** – See signal.plot docstring for full description

• **navigator_list** – 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** – Any extra keyword arguments are passed to each signal plot method.

**Example**

```python
>>> s_cl = hs.load("coreloss.dm3")
>>> s_ll = hs.load("lowloss.dm3")
>>> hs.plot.plot_signals([s_cl, s_ll])
```

Specifying the navigator:

```python
>>> s_cl = hs.load("coreloss.dm3")
>>> s_ll = hs.load("lowloss.dm3")
>>> hs.plot.plot_signals([s_cl, s_ll], navigator="slider")
```

Specifying the navigator for each signal:

```python
>>> s_cl = hs.load("coreloss.dm3")
>>> s_ll = hs.load("lowloss.dm3")
>>> s_edx = hs.load("edx.dm3")
>>> s_adf = hs.load("adf.dm3")
>>> hs.plot.plot_signals([s_cl, s_ll, s_edx], navigator_list=["slider",None,s_adf])
```

 hyperspy.drawing.utils.plot_spectra(spectra, style='overlap', color=None, line_style=None, padding=1.0, legend=None, legend_picking=True, legend_loc='upper right', fig=None, ax=None, **kwargs)

Plot several spectra in the same figure.

Extra keyword arguments are passed to matplotlib.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") – The style of the plot.

• **color** (matplotlib color or a list of them or None) – Sets the color of the lines of the plots (no action on ‘heatmap’). If a list, its length is less than the number of spectra to plot, the colors will be cycled. If None, use default matplotlib color cycle.

• **line_style** (matplotlib line style or a list of them or None) – Sets the line style of the plots (no action on ‘heatmap’). The main line style are ‘-’, ‘--’, ‘steps’, ‘:’, ‘.’. If a list, its length is less than the number of spectra to plot, line_style will be cycled. If None, use continuous lines, e.g: (‘-’, ‘--’, ‘steps’, ‘:’, ‘.’)

• **padding** (float, optional, default 0.1) – Option for “cascade”. 1 guarantees that there is not 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 or list of str or 'auto')* – 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.

- **legend_picking** *(bool)* – If true, a spectrum can be toggle on and off by clicking on
  the legended line.

- **legend_loc** *(str or int)* – This parameter controls where the legend is placed on
  the figure; see the pyplot.legend docstring for valid values

- **fig** *(matplotlib figure or None)* – If None, a default figure will be created.
  Specifying fig will not work for the ‘heatmap’ style.

- **ax** *(matplotlib ax (subplot) or None)* – If None, a default ax will be created.
  Will not work for ‘mosaic’ or ‘heatmap’ style.

- ****kwargs** – remaining keyword arguments are passed to matplotlib.figure() or
  matplotlib.subplots(). Has no effect on ‘heatmap’ style.

**Example**

```python
>>> s = hs.load("some_spectra")
>>> hs.plot.plot_spectra(s, style='cascade', color='red', padding=0.5)
```

To save the plot as a png-file

```python
>>> hs.plot.plot_spectra(s).figure.savefig("test.png")
```

**Returns**

- **ax** – An array is returned when *style* is “mosaic”.

**Return type**

matplotlib axes or list of matplotlib axes

**hyperspy.drawing.utils.set_axes_decor(ax, axes_decor)**

**hyperspy.drawing.utils.set_xaxis_lims(mpl_ax, hs_axis)**

Set the matplotlib axis limits to match that of a HyperSpy axis

**Parameters**

- **mpl_ax** *(matplotlib.axis.Axis)* – The matplotlib axis to change

- **hs_axis** *(DataAxis)* – The data axis that contains the values that control the scaling

**hyperspy.drawing.utils.subplot_parameters(fig)**

Returns a list of the subplot parameters of a mpl figure.

**Parameters**

- **fig** *(mpl figure)* –

**Returns**

- **tuple**

**Return type**

(left, bottom, right, top, wspace, hspace)

**hyperspy.drawing.widget module**

**class hyperspy.drawing.widget.DraggableWidgetBase(axes_manager, **kwargs)**

**Bases:** hyperspy.drawing.widget.WidgetBase
HyperSpy Documentation, Release 1.5.1.dev

Adds the position and indices properties, and adds a framework for letting the user drag the patch around. Also adds the moved event.

The default behavior is that position snaps to the values corresponding to the values of the axes grid (i.e. no subpixel values). This behavior can be controlled by the property snap_position.

Any inheritors must override these methods: _onmousemove(self, event) _update_patch_position(self) _set_patch(self)

button_release(event)
whenever a mouse button is released

connect(ax)
Connect to the matplotlib Axes’ events.

property indices
onpick(event)
property position
property snap_position

class hyperspy.drawing.widget.ResizableDraggableWidgetBase(axes_manager, **kwargs)
Bases: hyperspy.drawing.widget.DraggableWidgetBase

Adds the size property and get_size_in_axes method, and adds a framework for letting the user resize the patch, including resizing by key strokes (+, -). Also adds the ‘resized’ event.

Utility functions for resizing are implemented by increase_size and decrease_size, which will in-/decrement the size by 1. Other utility functions include get_centre and get_centre_indices which returns the center position, and the internal _apply_changes which helps make sure that only one ‘changed’ event is fired for a combined move and resize.

Any inheritors must override these methods: _update_patch_position(self) _update_patch_size(self) _update_patch_geometry(self) _set_patch(self)

button_release(event)
whenever a mouse button is released

connect(ax)
Connect to the matplotlib Axes’ events.

decrease_size()
Decrement all sizes by 1. Applied via ‘size’ property.

get_centre()
Get’s the center indices. The default implementation is simply the position + half the size in axes space, which should work for any symmetric widget, but more advanced widgets will need to decide whether to return the center of gravity or the geometrical center of the bounds.

get_centre_index()
Get’s the center position (in index space). The default implementation is simply the indices + half the size, which should work for any symmetric widget, but more advanced widgets will need to decide whether to return the center of gravity or the geometrical center of the bounds.

get_size_in_indices()
Gets the size property converted to the index space (via ‘axes’ attribute).

increase_size()
Increment all sizes by 1. Applied via ‘size’ property.

on_key_press(event)

3.1. hyperspy package
onpick (event)

set_size_in_indices (value)
Sets the size property converted to the index space (via 'axes' attribute).

property size
property snap_all
property snap_size

class hyperspy.drawing.widget.ResizersMixin (resizers=True, **kwargs)
Bases: object

Widget mix-in for adding resizing manipulation handles.
The default handles are green boxes displayed on the outside corners of the boundaries. By default, the handles
are only displayed when the widget is selected (picked in matplotlib terminology).

resizers [[bool]] Property that determines whether the resizer handles should be used

resize_color [[matplotlib color]] The color of the resize handles.

resize_pixel_size [[tuple | None]] Size of the resize handles in screen pixels. If None, it is set equal
to the size of one ‘data-pixel’ (image pixel size).

resizer_picked [[False | int]] Indicates which, if any, resizer was selected the last time the widget
was picked. False if another patch was picked, or the index of the resizer handle that was picked.

onpick (event)
Picking of main patch is same as for widget base, but this also handles picking of the resize handles. If
a resize handles is picked, picked is set to True, and resizer_picked is set to an integer indicating which
handle was picked (0-3 for top left, top right, bottom left, bottom right). It is set to False if another widget
was picked.

If the main patch is picked, the offset from the picked pixel to the position is stored in pick_offset. This can
be used in e.g. _onmousemove to ease dragging code (prevent widget center/corner snapping to mouse).

property resizers

set_on (value)
Turns on/off resizers what widget is turned on/off.

class hyperspy.drawing.widget.Widget1DBase (axes_manager, **kwargs)
Bases: hyperspy.drawing.widget.DraggableWidgetBase

A base class for 1D widgets.

It sets the right dimensions for size and position, adds the ‘border_thickness’ attribute and initializes the ‘axes’
attribute to the first two navigation axes if possible, if not, the two first signal_axes are used. Other than that it
mainly supplies common utility functions for inheritors, and implements required functions for ResizableDrag-
gableWidgetBase.

The implementation for ResizableDraggableWidgetBase methods all assume that a Rectangle patch will be used,
centered on position. If not, the inheriting class will have to override those as applicable.

class hyperspy.drawing.widget.Widget2DBase (axes_manager, **kwargs)
Bases: hyperspy.drawing.widget.ResizableDraggableWidgetBase

A base class for 2D widgets. It sets the right dimensions for size and position, adds the ‘border_thickness’
attribute and initializes the ‘axes’ attribute to the first two navigation axes if possible, if not, the two first sig-
nal_axes are used. Other than that it mainly supplies common utility functions for inheritors, and implements
required functions for ResizableDraggableWidgetBase.
The implementation for ResizableDraggableWidgetBase methods all assume that a Rectangle patch will be used, centered on position. If not, the inheriting class will have to override those as applicable.

class hyperspy.drawing.widget.WidgetBase(axes_manager=None, color='red', alpha=1.0, **kwargs)

Bases: object

Base class for interactive widgets/patches. A widget creates and maintains one or more matplotlib patches, and manages the interaction code so that the user can manipulate it on the fly.

This base class implements functionality which is common to all such widgets, mainly the code that manages the patch, axes management, and sets up common events (‘changed’ and ‘closed’).

Any inheriting subclasses must implement the following methods: _set_patch(self) _on_navigate(obj, name, old, new) # Only for widgets that can navigate

It should also make sure to fill the ‘axes’ attribute as early as possible (but after the base class init), so that it is available when needed.

property alpha

property axes

close(window=None)

Set the on state to off (removes patch and disconnects), and trigger events.closed.

property color

connect(ax)

Connect to the matplotlib Axes’ events.

connect_navigate()

Connect to the axes_manager such that changes in the widget or in the axes_manager are reflected in the other.

disconnect()

Disconnect from all events (both matplotlib and navigation).

disconnect_navigate()

Disconnect a previous navigation connection.

draw_patch(*args)

Update the patch drawing.

is_on()

Determines if the widget is set to draw if valid (turned on).

select()

Cause this widget to be the selected widget in its MPL axes. This assumes that the widget has its patch added to the MPL axes.

set_mpl_ax(ax)

Set the matplotlib Axes that the widget will draw to. If the widget on state is True, it will also add the patch to the Axes, and connect to its default events.

set_on(value)

Change the on state of the widget. If turning off, all patches will be removed from the matplotlib axes and the widget will disconnect from all events. If turning on, the patch(es) will be added to the matplotlib axes, and the widget will connect to its default events.
hyperspy.drawing.widgets module

Interactive widgets that can be added to Signal plots.

Example

Module contents

hyperspy.external package

Subpackages

hyperspy.external.astroML package

Submodules

hyperspy.external.astroML.bayesian_blocks module

Bayesian Block implementation

Dynamic programming algorithm for finding the optimal adaptive-width histogram.
Based on Scargle et al 2012 [1].

References

class hyperspy.external.astroML.bayesian_blocks.Events(p0=0.05, gamma=None)
Bases: hyperspy.external.astroML.bayesian_blocks.FitnessFunc

Fitness for binned or unbinned events

Parameters

- p0 (float) – False alarm probability, used to compute the prior on N (see eq. 21 of Scargle 2012). Default prior is for p0 = 0.
- gamma (float or None) – If specified, then use this gamma to compute the general prior form, p ~ gamma^N. If gamma is specified, p0 is ignored.

fitness(N_k, T_k)

prior(N, Ntot)

class hyperspy.external.astroML.bayesian_blocks.FitnessFunc(p0=0.05, gamma=None)
Bases: object

Base class for fitness functions
Each fitness function class has the following: - fitness(...) : compute fitness function.
Arguments accepted by fitness must be among [T_k, N_k, a_k, b_k, c_k]

- prior(N, Ntot) : compute prior on N given a total number of points Ntot

property args
fitness()

gamma_prior \((N, N_{tot})\)
Basic prior, parametrized by gamma (eq. 3 in Scargle 2012)

p0_prior \((N, N_{tot})\)

prior \((N, N_{tot})\)

validate_input \((t, x, \sigma)\)
Check that input is valid

class hyperspy.external.astroML.bayesian_blocks.PointMeasures \((p0=None,\gamma=None)\)
Bases: hyperspy.external.astroML.bayesian_blocks.FitnessFunc
Fit by point measures

Parameters

- **gamma** *(float)* – specifies the prior on the number of bins: \(p \sim \gamma^N\) if \(\gamma\) is not specified, then a prior based on simulations will be used (see sec 3.3 of Scargle 2012)

fitness \((a_k, b_k)\)

prior \((N, N_{tot})\)

class hyperspy.external.astroML.bayesian_blocks.RegularEvents \((dt, p0=0.05, \gamma=None)\)
Bases: hyperspy.external.astroML.bayesian_blocks.FitnessFunc
Fitness for regular events

This is for data which has a fundamental “tick” length, so that all measured values are multiples of this tick length. In each tick, there are either zero or one counts.

Parameters

- **dt** *(float)* – tick rate for data
- **gamma** *(float)* – specifies the prior on the number of bins: \(p \sim \gamma^N\)

fitness \((T_k, N_k)\)

validate_input \((t, x, \sigma)\)
Check that input is valid

hyperspy.external.astroML.bayesian_blocks.bayesian_blocks \((t, x=None, \sigma=None, fitness='events', **kwargs)\)

Bayesian Blocks Implementation

This is a flexible implementation of the Bayesian Blocks algorithm described in Scargle 2012 [1].

Parameters

- **t** *(array_like)* – data times (one dimensional, length \(N\))
- **x** *(array_like (optional)) – data values*
- **sigma** *(array_like or float (optional)) – data errors*
- **fitness** *(str or object)* – the fitness function to use. If a string, the following options are supported:
  - ’events’ [binned or unbinned event data] extra arguments are \(p0\), which gives the false alarm probability to compute the prior, or \(\gamma\) which gives the slope of the prior on the number of bins.
- **'regular_events'** [non-overlapping events measured at multiples] of a fundamental tick rate, $dt$, which must be specified as an additional argument. The prior can be specified through $gamma$, which gives the slope of the prior on the number of bins.

- **'measures'** [fitness for a measured sequence with Gaussian errors] The prior can be specified using $gamma$, which gives the slope of the prior on the number of bins. If $gamma$ is not specified, then a simulation-derived prior will be used.

Alternatively, the fitness can be a user-specified object of type derived from the FitnessFunc class.

**Returns edges** – array containing the (N+1) bin edges

**Return type** ndarray

### Examples

Event data:

```python
t = np.random.normal(size=100)
bins = bayesian_blocks(t, fitness='events', p0=0.01)
```

Event data with repeats:

```python
t = np.random.normal(size=100)
t[80:] = t[:20]
bins = bayesian_blocks(t, fitness='events', p0=0.01)
```

Regular event data:

```python
dt = 0.01
t = dt * np.arange(1000)
x = np.zeros(len(t))
x[np.random.randint(0, len(t), len(t) / 10)] = 1
bins = bayesian_blocks(t, fitness='regular_events', dt=dt, gamma=0.9)
```

Measured point data with errors:

```python
t = 100 * np.random.random(100)
x = np.exp(-0.5 * (t - 50) ** 2)
sigma = 0.1
x_obs = np.random.normal(x, sigma)
bins = bayesian_blocks(t, fitness='measures')
```

### References

**See also:**

`astroml.plotting.hist()` histogram plotting function which can make use of bayesian blocks.

`hyperspy.external.astroml.histtools` module

Tools for working with distributions
**class** hyperspy.external.astroML.histtools.KnuthF*(data)*

Bases: object

Class which implements the function minimized by knuth_bin_width

**Parameters**

data (array-like, one dimension) – data to be histogrammed

**Notes**

the function $F$ is given by

$$F(M|x, I) = n \log(M) + \log \Gamma\left(\frac{M}{2}\right) - M \log \Gamma\left(\frac{1}{2}\right) - \log \Gamma\left(\frac{2n + M}{2}\right) + \sum_{k=1}^{M} \log \Gamma(n_k + \frac{1}{2})$$

where $\Gamma$ is the Gamma function, $n$ is the number of data points, $n_k$ is the number of measurements in bin $k$.

See also:

*knuth_bin_width*, *astroML.plotting.hist*

**bins**(M)

Return the bin edges given a width $dx$

**eval**(M)

Evaluate the Knuth function

**Parameters**

dx (float) – Width of bins

**Returns**

$F$ – evaluation of the negative Knuth likelihood function: smaller values indicate a better fit.

**Return type** float

**hyperspy.external.astroML.histtools.dasky_freedman_bin_width**(data, return_bins=True)

Dask version of freedman_bin_width

**Parameters**

- data (dask array) – the data
- return_bins (bool (optional)) – if True, then return the bin edges

**Returns**

- width (float) – optimal bin width using Scott’s rule
- bins (ndarray) – bin edges: returned if return_bins is True

**Notes**

The optimal bin width is

$$\Delta_b = \frac{2(q_{75} - q_{25})}{n^{1/3}}$$

where $q_N$ is the $N$ percent quartile of the data, and $n$ is the number of data points.

See also:

*knuth_bin_width(), scotts_bin_width(), astroML.plotting.hist()*
hyperspy.external.astroML.histtools.dasky_histogram(a, bins=10, **kwargs)
Enhanced histogram for dask arrays. The range keyword is ignored. Reads the data at most two times - once to determine best bins (if required), and second time to actually calculate the histogram.

Parameters

- **a** (array_like) – array of data to be histogrammed
- **bins** (int or list or str (optional)) – If bins is a string, then it must be one of: ‘scotts’ : use Scott’s rule to determine bins ‘freedman’ : use the Freedman-Diaconis rule to determine bins
- **keyword arguments are described in numpy.hist() (other)** –

Returns

- hist (array) – The values of the histogram. See normed and weights for a description of the possible semantics.
- bin_edges (array of dtype float) – Return the bin edges (length(hist)+1).

See also:

numpy.histogram(), astroML.plotting.hist()

hyperspy.external.astroML.histtools.dasky_scotts_bin_width(data, return_bins=True)
Dask version of scotts_bin_width

Parameters

- **data** (dask array) – the data
- **return_bins** (bool (optional)) – if True, then return the bin edges

Returns

- width (float) – optimal bin width using Scott’s rule
- bins (ndarray) – bin edges: returned if return_bins is True

Notes

The optimal bin width is:

$$
\Delta b = \frac{3.5\sigma}{n^{1/3}}
$$

where $\sigma$ is the standard deviation of the data, and $n$ is the number of data points.

See also:

knuth_bin_width(), freedman_bin_width(), astroML.plotting.hist()

hyperspy.external.astroML.histtools.freedman_bin_width(data, return_bins=False)
Return the optimal histogram bin width using the Freedman-Diaconis rule

Parameters

- **data** (array-like, ndim=1) – observed (one-dimensional) data
- **return_bins** (bool (optional)) – if True, then return the bin edges

Returns

- width (float) – optimal bin width using Scott’s rule
• **bins** *(ndarray)* – bin edges: returned if **return_bins** is True

**Notes**

The optimal bin width is

$$\Delta_b = \frac{2(q_{75} - q_{25})}{n^{1/3}}$$

where \(q_N\) is the \(N\) percent quartile of the data, and \(n\) is the number of data points.

**See also:**

*knuth_bin_width(), scotts_bin_width(), astroML.plotting.hist()*

**hyperspy.external.astroML.histtools.histogram(a, bins=10, range=None, **kwargs)**

Enhanced histogram

This is a histogram function that enables the use of more sophisticated algorithms for determining bins. Aside from the **bins** argument allowing a string specified how bins are computed, the parameters are the same as numpy.histogram().

**Parameters**

- **a** *(array_like)* – array of data to be histogrammed
- **bins** *(int or list or str (optional)) – If bins is a string, then it must be one of: ‘blocks’ : use bayesian blocks for dynamic bin widths ‘knuth’ : use Knuth’s rule to determine bins ‘scotts’ : use Scott’s rule to determine bins ‘freedman’ : use the Freedman-Diaconis rule to determine bins*
- **range** *(tuple or None (optional)) – the minimum and maximum range for the histogram. If not specified, it will be (x.min(), x.max())*
- **keyword arguments are described in numpy.histogram() (other)** –

**Returns**

- **hist** *(array)* – The values of the histogram. See **normed** and **weights** for a description of the possible semantics.
- **bin_edges** *(array of dtype float)* – Return the bin edges \((\text{length(hist)} + 1)\).

**See also:**

*numpy.histogram(), astroML.plotting.hist()*

**hyperspy.external.astroML.histtools.knuth_bin_width(data, return_bins=False)**

Return the optimal histogram bin width using Knuth’s rule¹

**Parameters**

- **data** *(array-like, ndim=1)* – observed (one-dimensional) data
- **return_bins** *(bool (optional)) – if True, then return the bin edges*

**Returns**

- **dx** *(float)* – optimal bin width. Bins are measured starting at the first data point.
- **bins** *(ndarray)* – bin edges: returned if **return_bins** is True

Notes

The optimal number of bins is the value \( M \) which maximizes the function

\[
F(M|x, I) = n \log(M) + \log \Gamma\left(\frac{M}{2}\right) - M \log \Gamma\left(\frac{1}{2}\right) - \log \Gamma\left(\frac{2n + M}{2}\right) + \sum_{k=1}^{M} \log \Gamma(n_k + \frac{1}{2})
\]

where \( \Gamma \) is the Gamma function, \( n \) is the number of data points, \( n_k \) is the number of measurements in bin \( k \).

References

See also:

KnuthF(), freedman_bin_width(), scotts_bin_width()

 hyperspy.external.astroML.histtools.scotts_bin_width(data, return_bins=False)

Return the optimal histogram bin width using Scott’s rule:

**Parameters**

- **data** (array-like, ndim=1) – observed (one-dimensional) data
- **return_bins** (bool (optional)) – if True, then return the bin edges

**Returns**

- **width** (float) – optimal bin width using Scott’s rule
- **bins** (ndarray) – bin edges: returned if **return_bins** is True

Notes

The optimal bin width is

\[
\Delta_b = \frac{3.5\sigma}{n^{1/3}}
\]

where \( \sigma \) is the standard deviation of the data, and \( n \) is the number of data points.

See also:

knuth_bin_width(), freedman_bin_width(), astroML.plotting.hist()
Craig Markwardt converted the FORTRAN code to IDL. The information for the IDL version is:

Craig B. Markwardt, NASA/GSFC Code 662, Greenbelt, MD 20770
craigm@lheamail.gsfc.nasa.gov UPDATED VERSIONs can be found on my WEB PAGE:
http://cow.physics.wisc.edu/~craigm/idl/idl.html

Mark Rivers created this Python version from Craig’s IDL version. Mark Rivers, University of Chicago Building 434A, Argonne National Laboratory 9700 South Cass Avenue, Argonne, IL 60439 rivers@cars.uchicago.edu Updated versions can be found at http://cars.uchicago.edu/software

Sergey Koposov converted the Mark’s Python version from Numeric to numpy
Sergey Koposov, University of Cambridge, Institute of Astronomy, Madingley road, CB3 0HA, Cambridge, UK koposov@ast.cam.ac.uk Updated versions can be found at http://code.google.com/p/astrolibpy/source/browse/trunk/

DESCRIPTION

MPFIT uses the Levenberg-Marquardt technique to solve the least-squares problem. In its typical use, MPFIT will be used to fit a user-supplied function (the “model”) to user-supplied data points (the “data”) by adjusting a set of parameters. MPFIT is based upon MINPACK-1 (LMDIF.F) by More’ and collaborators.

For example, a researcher may think that a set of observed data points is best modelled with a Gaussian curve. A Gaussian curve is parameterized by its mean, standard deviation and normalization. MPFIT will, within certain constraints, find the set of parameters which best fits the data. The fit is “best” in the least-squares sense; that is, the sum of the weighted squared differences between the model and data is minimized.

The Levenberg-Marquardt technique is a particular strategy for iteratively searching for the best fit. This particular implementation is drawn from MINPACK-1 (see NETLIB), and is much faster and more accurate than the version provided in the Scientific Python package in Scientific.Functions.LeastSquares. This version allows upper and lower bounding constraints to be placed on each parameter, or the parameter can be held fixed.

The user-supplied Python function should return an array of weighted deviations between model and data. In a typical scientific problem the residuals should be weighted so that each deviate has a gaussian sigma of 1.0. If X represents values of the independent variable, Y represents a measurement for each value of X, and ERR represents the error in the measurements, then the deviates could be calculated as follows:

\[ \text{DEVIATES} = \frac{(Y - F(X))}{\text{ERR}} \]

where F is the analytical function representing the model. You are recommended to use the convenience functions MPFITFUN and MPFITEXPR, which are driver functions that calculate the deviates for you. If ERR are the 1-sigma uncertainties in Y, then

\[ \text{TOTAL} (\text{DEVIATES}^2) \]

will be the total chi-squared value. MPFIT will minimize the chi-square value. The values of X, Y and ERR are passed through MPFIT to the user-supplied function via the FUNCTKW keyword.

Simple constraints can be placed on parameter values by using the PARINFO keyword to MPFIT. See below for a description of this keyword.

MPFIT does not perform more general optimization tasks. See TNMIN instead. MPFIT is customized, based on MINPACK-1, to the least-squares minimization problem.

USER FUNCTION
The user must define a function which returns the appropriate values as specified above. The function should return the weighted deviations between the model and the data. It should also return a status flag and an optional partial derivative array. For applications which use finite-difference derivatives – the default – the user function should be declared in the following way:

```python
def myfunct(p, fjac=None, x=None, y=None, err=None) # Parameter values are passed in “p” # If fjac==None then partial derivatives should not be # computed. It will always be None if MPFIT is called with default # flag. model = F(x, p) # Non-negative status value means MPFIT should continue, negative means # stop the calculation. status = 0 return([status, (y-model)/err]
```

See below for applications with analytical derivatives.

The keyword parameters X, Y, and ERR in the example above are suggestive but not required. Any parameters can be passed to MYFUNCT by using the functkw keyword to MPFIT. Use MPFITFUN and MPFITEXPR if you need ideas on how to do that. The function must accept a parameter list, P.

In general there are no restrictions on the number of dimensions in X, Y or ERR. However the deviates must be returned in a one-dimensional Numeric array of type Float.

User functions may also indicate a fatal error condition using the status return described above. If status is set to a number between -15 and -1 then MPFIT will stop the calculation and return to the caller.

### ANALYTIC DERIVATIVES

In the search for the best-fit solution, MPFIT by default calculates derivatives numerically via a finite difference approximation. The user-supplied function need not calculate the derivatives explicitly. However, if you desire to compute them analytically, then the AUTODERIVATIVE=0 keyword must be passed to MPFIT. As a practical matter, it is often sufficient and even faster to allow MPFIT to calculate the derivatives numerically, and so AUTODERIVATIVE=0 is not necessary.

If AUTODERIVATIVE=0 is used then the user function must check the parameter FJAC, and if FJAC!=None then return the partial derivative array in the return list.

```python
def myfunct(p, fjac=None, x=None, y=None, err=None) # Parameter values are passed in “p” # If FJAC!=None then partial derivatives must be computed. # FJAC contains an array of len(p), where each entry # is 1 if that parameter is free and 0 if it is fixed. model = F(x, p) # Non-negative status value means MPFIT should continue, negative means # stop the calculation. status = 0 if (dojac):
    pderiv = zeros([len(x), len(p)], Float) for j in range(len(p)):
        pderiv[:,j] = FGRAD(x, p, j)
else:
    pderiv = None
return([status, (y-model)/err, pderiv]
```

where FGRAD(x, p, i) is a user function which must compute the derivative of the model with respect to parameter P[i] at X. When finite differencing is used for computing derivatives (i.e., when AUTODERIVATIVE=1), or when MPFIT needs only the errors but not the derivatives the parameter FJAC=None.

Derivatives should be returned in the PDERIV array. PDERIV should be an m x n array, where m is the number of data points and n is the number of parameters. dp[i,j] is the derivative at the ith point with respect to the jth parameter.

The derivatives with respect to fixed parameters are ignored; zero is an appropriate value to insert for those derivatives. Upon input to the user function, FJAC is set to a vector with the same length as P, with a value of 1 for a parameter which is free, and a value of zero for a parameter which is fixed (and hence no derivative needs to be calculated).
If the data is higher than one dimensional, then the last dimension should be the parameter dimension. Example: fitting a 50x50 image, “dp” should be 50x50xNPAR.

CONSTRaining PARAMETER VALUES WITH THE PARINFO KEYWORD

The behavior of MPFIT can be modified with respect to each parameter to be fitted. A parameter value can be fixed; simple boundary constraints can be imposed; limitations on the parameter changes can be imposed; properties of the automatic derivative can be modified; and parameters can be tied to one another.

These properties are governed by the PARINFO structure, which is passed as a keyword parameter to MPFIT.

PARINFO should be a list of dictionaries, one list entry for each parameter. Each parameter is associated with one element of the array, in numerical order. The dictionary can have the following keys (none are required, keys are case insensitive):

- 'value' - the starting parameter value (but see the START_PARAMS parameter for more information).
- 'fixed' - a boolean value, whether the parameter is to be held fixed or not. Fixed parameters are not varied by MPFIT, but are passed on to MYFUNCT for evaluation.
- 'limited' - a two-element boolean array. If the first/second element is set, then the parameter is bounded on the lower/upper side. A parameter can be bounded on both sides. Both LIMITED and LIMITS must be given together.
- 'limits' - a two-element float array. Gives the parameter limits on the lower and upper sides, respectively. Zero, one or two of these values can be set, depending on the values of LIMITED. Both LIMITED and LIMITS must be given together.
- 'parname' - a string, giving the name of the parameter. The fitting code of MPFIT does not use this tag in any way. However, the default iterfunct will print the parameter name if available.
- 'step' - the step size to be used in calculating the numerical derivatives. If set to zero, then the step size is computed automatically. Ignored when AUTODERIVATIVE=0.
- 'mpside' - the sidedness of the finite difference when computing numerical derivatives. This field can take four values:
  - 0 - one-sided derivative computed automatically
  - 1 - one-sided derivative \((f(x+h) - f(x))/h\)
  - -1 - one-sided derivative \((f(x) - f(x-h))/h\)
  - 2 - two-sided derivative \((f(x+h) - f(x-h))/(2*h)\)

Where \(H\) is the STEP parameter described above. The “automatic” one-sided derivative method will choose a direction for the finite difference which does not violate any constraints. The other methods do not perform this check. The two-sided method is in principle more precise, but requires twice as many function evaluations. Default: 0.

- 'mpmaxstep' - the maximum change to be made in the parameter value. During the fitting process, the parameter will never be changed by more than this value in one iteration.
  
  A value of 0 indicates no maximum. Default: 0.

- 'tied' - a string expression which “ties” the parameter to other free or fixed parameters. Any expression involving constants and the parameter array \(P\) are permitted. Example: if parameter 2 is always to be twice parameter 1 then use the following: parinfo(2).tied = ‘2 * p(1)’. Since they are totally constrained, tied parameters are considered to be fixed; no errors are computed for them. [ NOTE: the PARNAME can’t be used in expressions. ]
‘mpprint’ - if set to 1, then the default iterfunct will print the parameter value. If set to 0, the parameter value will not be printed. This tag can be used to selectively print only a few parameter values out of many. Default: 1 (all parameters printed)

Future modifications to the PARINFO structure, if any, will involve adding dictionary tags beginning with the two letters “MP”. Therefore programmers are urged to avoid using tags starting with the same letters; otherwise they are free to include their own fields within the PARINFO structure, and they will be ignored.

PARINFO Example: parinfo = [{‘value’:0., ‘fixed’:0, ‘limited’:[0,0], ‘limits’:[0.,0.,0.]}]

for i in range(5)
parinfo[0][‘fixed’] = 1 parinfo[4][‘limited’][0] = 1 parinfo[4][‘limits’][0] = 50. values = [5.7, 2.2, 500., 1.5, 2000.]
for i in range(5): parinfo[i][‘value’]=values[i]

A total of 5 parameters, with starting values of 5.7, 2.2, 500, 1.5, and 2000 are given. The first parameter is fixed at a value of 5.7, and the last parameter is constrained to be above 50.

EXAMPLE

import mpfit import numpy.oldnumeric as Numeric x = arange(100, float) p0 = [5.7, 2.2, 500., 1.5, 2000.]
for i in range(5)
y = ( p[0] + p[1]*[x] + p[2]*[x**2] + p[3]*sqrt(x) + p[4]*log(x))
fa = {‘x’:x, ‘y’:y, ‘err’:err} m = mpfit(‘myfunct’, p0, functkw=fa)
print(‘status = ‘, m.status) if (m.status <= 0):
print(‘error message = ‘, m.errmsg)
print(‘parameters = ‘, m.params)

Minimizes sum of squares of MYFUNCT. MYFUNCT is called with the X, Y, and ERR keyword parameters that are given by FUNCTKW. The results can be obtained from the returned object m.

THEORY OF OPERATION

There are many specific strategies for function minimization. One very popular technique is to use function gradient information to realize the local structure of the function. Near a local minimum the function value can be taylor expanded about x0 as follows:

\[
f(x) = f(x_0) + f'(x_0) \cdot (x-x_0) + \frac{1}{2} (x-x_0) \cdot f''(x_0) \cdot (x-x_0)
\]  
—–  

Order 0th 1st 2nd

Here \( f'(x) \) is the gradient vector of \( f \) at \( x \), and \( f''(x) \) is the Hessian matrix of second derivatives of \( f \) at \( x \). The vector \( x \) is the set of function parameters, not the measured data vector. One can find the minimum of \( f \), \( f(xm) \) using Newton’s method, and arrives at the following linear equation:

\[
f''(x_0) \cdot (xm-x_0) = -f'(x_0)
\]  

If an inverse can be found for \( f''(x_0) \) then one can solve for \( (xm-x_0) \), the step vector from the current position \( x_0 \) to the new projected minimum. Here the problem has been linearized (ie, the gradient information is known to first order). \( f''(x_0) \) is symmetric \( n \times n \) matrix, and should be positive definite.

The Levenberg - Marquardt technique is a variation on this theme. It adds an additional diagonal term to the equation which may aid the convergence properties:

\[
(f''(x_0) + nu I) \cdot (xm-x_0) = -f'(x_0)
\]  

where \( I \) is the identity matrix. When \( nu \) is large, the overall matrix is diagonally dominant, and the iterations follow steepest descent. When \( nu \) is small, the iterations are quadratically convergent.
In principle, if \( f''(x_0) \) and \( f'(x_0) \) are known then \( x_m - x_0 \) can be determined. However the Hessian matrix is often difficult or impossible to compute. The gradient \( f'(x_0) \) may be easier to compute, if even by finite difference techniques. So-called quasi-Newton techniques attempt to successively estimate \( f''(x_0) \) by building up gradient information as the iterations proceed.

In the least squares problem there are further simplifications which assist in solving eqn (2). The function to be minimized is a sum of squares:

\[
    f = \sum(h_i^2) \quad (3)
\]

where \( h_i \) is the \( i \)th residual out of \( m \) residuals as described above. This can be substituted back into eqn (2) after computing the derivatives:

\[
    f' = 2 \sum(h_i h_i') \quad f'' = 2 \sum(h_i' h_j') + 2 \sum(h_i h_i'') \quad (4)
\]

If one assumes that the parameters are already close enough to a minimum, then one typically finds that the second term in \( f'' \) is negligible [or, in any case, is too difficult to compute]. Thus, equation (2) can be solved, at least approximately, using only gradient information.

In matrix notation, the combination of eqns (2) and (4) becomes:

\[
    h^T' \cdot h' \cdot dx = - h^T' \cdot h \quad (5)
\]

where \( h \) is the residual vector (length \( m \)), \( h^T \) is its transpose, \( h' \) is the Jacobian matrix (dimensions \( n \times m \)), and \( dx \) is \( (x_m - x_0) \). The user function supplies the residual vector \( h \), and in some cases \( h' \) when it is not found by finite differences (see MPFIT_FDJAC2, which finds \( h \) and \( h^T' \)). Even if \( dx \) is not the best absolute step to take, it does provide a good estimate of the best direction, so often a line minimization will occur along the \( dx \) vector direction.

The method of solution employed by MINPACK is to form the \( Q \cdot R \) factorization of \( h' \), where \( Q \) is an orthogonal matrix such that \( QT \cdot Q = I \), and \( R \) is upper right triangular. Using \( h' = Q \cdot R \) and the orthogonality of \( Q \), eqn (5) becomes

\[
    (RT \cdot QT) \cdot (Q \cdot R) \cdot dx = - (RT \cdot QT) \cdot h \quad (6)
\]

\[
    RT \cdot R \cdot dx = - RT \cdot QT \cdot h \quad (6)
\]

where the last statement follows because \( R \) is upper triangular. Here, \( R \), \( QT \) and \( h \) are known so this is a matter of solving for \( dx \). The routine MPFIT_QRFAC provides the QR factorization of \( h \), with pivoting, and MPFIT_QRSOLV provides the solution for \( dx \).

REFERENCES


MODIFICATION HISTORY

Translated from MINPACK-1 in FORTRAN, Apr-Jul 1998, CM

Copyright (C) 1997-2002, Craig Markwardt This software is provided as is without any warranty whatsoever. Permission to use, copy, modify, and distribute modified or unmodified copies is granted, provided this copyright and disclaimer are included unchanged.

Translated from MPFIT (Craig Markwardt’s IDL package) to Python, August, 2002. Mark Rivers Converted from Numeric to numpy (Sergey Koposov, July 2008) Fixed the analytic derivatives features (The HyperSpy Developers, 2011)
**class** hyperspy.external.mpfit.mpfit.machar

Bases: object

**class** hyperspy.external.mpfit.mpfit

Bases: object

**Inputs:**

**fcn:**

The function to be minimized. The function should return the weighted deviations between the model and the data, as described above.

**xall:**

An array of starting values for each of the parameters of the model. The number of parameters should be fewer than the number of measurements.

This parameter is optional if the parinfo keyword is used (but see parinfo). The parinfo keyword provides a mechanism to fix or constrain individual parameters.

**Keywords:**

**autoderivative:** If this is set, derivatives of the function will be computed automatically via a finite differencing procedure. If not set, then fcn must provide the (analytical) derivatives.

Default: set (=1) NOTE: to supply your own analytical derivatives, explicitly pass autoderivative=0

**ftol:** A nonnegative input variable. Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most ftol (and status is accordingly set to 1 or 3). Therefore, ftol measures the relative error desired in the sum of squares.

Default: 1E-10

**functkw:** A dictionary which contains the parameters to be passed to the user-supplied function specified by fcn via the standard Python keyword dictionary mechanism. This is the way you can pass additional data to your user-supplied function without using global variables.

Consider the following example:

```
if functkw = {'xval':[1.,2.,3.], 'yval':[1.,4.,9.], 'errval':[1.,1.,1.]} }

then the user supplied function should be declared like this: def myfunct(p, fjac=None, xval=None, yval=None, errval=None):

Default: {} No extra parameters are passed to the user-supplied function.
```

**gtol:** A nonnegative input variable. Termination occurs when the cosine of the angle between fvec and any column of the jacobian is at most gtol in absolute value (and status is accordingly set to 4). Therefore, gtol measures the orthogonality desired between the function vector and the columns of the jacobian.

Default: 1e-10

**iterkw:** The keyword arguments to be passed to iterfunc via the dictionary keyword mechanism. This should be a dictionary and is similar in operation to FUNCTKW.

Default: {} No arguments are passed.

**iterfunc:** The name of a function to be called upon each NPRINT iteration of the MPFIT routine. It should be declared in the following way:
def iterfunct(myfunct, p, iter, fnorm, functkw=None,
    parinfo=None, quiet=0,
    dof=None, [iterkw keywords here])

    # perform custom iteration update
    iterfunct must accept all three keyword parameters (FUNCTKW, PARINFO and QUIET).

myfunct: The user-supplied function to be minimized, p: The current set of model parameters
iter: The iteration number functkw: The arguments to be passed to myfunct. fnorm: The chi-
    squared value. quiet: Set when no textual output should be printed. dof: The number of degrees
    of freedom, normally the number of points

    less the number of free parameters.

See below for documentation of parinfo.

In implementation, iterfunct can perform updates to the terminal or graphical user interface, to
provide feedback while the fit proceeds. If the fit is to be stopped for any reason, then iterfunct
should return a a status value between -15 and -1. Otherwise it should return None (e.g. no
return statement) or 0. In principle, iterfunct should probably not modify the parameter values,
because it may interfere with the algorithm’s stability. In practice it is allowed.

Default: an internal routine is used to print the parameter values.

Set iterfunct=None if there is no user-defined routine and you don’t want the internal default
routine be called.

maxiter: The maximum number of iterations to perform. If the number is exceeded, then the status
value is set to 5 and MPFIT returns. Default: 200 iterations

nocovar: Set this keyword to prevent the calculation of the covariance matrix before returning (see
COVAR) Default: clear (=0) The covariance matrix is returned

nprint: The frequency with which iterfunct is called. A value of 1 indicates that iterfunct is called
with every iteration, while 2 indicates every other iteration, etc. Note that several Levenberg-
Marquardt attempts can be made in a single iteration. Default value: 1

parinfo Provides a mechanism for more sophisticated constraints to be placed on parameter values.
When parinfo is not passed, then it is assumed that all parameters are free and unconstrained.
Values in parinfo are never modified during a call to MPFIT.

See description above for the structure of PARINFO.

Default value: None All parameters are free and unconstrained.

quiet: Set this keyword when no textual output should be printed by MPFIT

damp: A scalar number, indicating the cut-off value of residuals where “damping” will occur. Resid-
uals with magnitudes greater than this number will be replaced by their hyperbolic tangent.
This partially mitigates the so-called large residual problem inherent in least-squares solvers (as
damping.

Default: 0

Note: DAMP doesn’t work with autoderivative=0

xtol: A nonnegative input variable. Termination occurs when the relative error between two consec-
utive iterates is at most xtol (and status is accordingly set to 2 or 3). Therefore, xtol measures
the relative error desired in the approximate solution. Default: 1E-10

Outputs:
Returns an object of type mpfit. The results are attributes of this class, e.g. mpfit.status, mpfit.errmsg, mpfit.params, mpfit.niter, mpfit.covar.

**.status** An integer status code is returned. All values greater than zero can represent success (however .status == 5 may indicate failure to converge). It can have one of the following values:

- **-16** A parameter or function value has become infinite or an undefined number. This is usually a consequence of numerical overflow in the user’s model function, which must be avoided.
- **-15 to -1** These are error codes that either MYFUNCT or iterfunct may return to terminate the fitting process. Values from -15 to -1 are reserved for the user functions and will not clash with MPFIT.

0 Improper input parameters.

1 Both actual and predicted relative reductions in the sum of squares are at most ftol.

2 Relative error between two consecutive iterates is at most xtol

3 Conditions for status = 1 and status = 2 both hold.

4 The cosine of the angle between fvec and any column of the jacobian is at most gtol in absolute value.

5 The maximum number of iterations has been reached.

6 ftol is too small. No further reduction in the sum of squares is possible.

7 xtol is too small. No further improvement in the approximate solution x is possible.

8 gtol is too small. fvec is orthogonal to the columns of the jacobian to machine precision.

**.fnorm** The value of the summed squared residuals for the returned parameter values.

**.covar** The covariance matrix for the set of parameters returned by MPFIT. The matrix is NxN where N is the number of parameters. The square root of the diagonal elements gives the formal 1-sigma statistical errors on the parameters if errors were treated “properly” in fcn. Parameter errors are also returned in .perror.

To compute the correlation matrix, pcor, use this example: cov = mpfit.covar pcor = cov * 0. for i in range(n):

    for j in range(n): pcor[i,j] = cov[i,j]/sqrt(cov[i,i]*cov[j,j])

If nocovar is set or MPFIT terminated abnormally, then .covar is set to a scalar with value None.

**.errmsg** A string error or warning message is returned.

**.nfev** The number of calls to MYFUNCT performed.

**.niter** The number of iterations completed.

**.perror** The formal 1-sigma errors in each parameter, computed from the covariance matrix. If a parameter is held fixed, or if it touches a boundary, then the error is reported as zero.

If the fit is unweighted (i.e. no errors were given, or the weights were uniformly set to unity), then .perror will probably not represent the true parameter uncertainties.

If you can assume that the true reduced chi-squared value is unity – meaning that the fit is implicitly assumed to be of good quality – then the estimated parameter uncertainties can be computed by scaling .perror by the measured chi-squared value.

    dof = len(x) - len(mpfit.params) # deg of freedom # scaled uncertainties perror = mpfit.perror * sqrt(mpfit.fnorm / dof)

```
blas_enorm32 = <fortran dnrm2>
```
blas_enorm64 = <fortran dnrm2>

calc_covar (rr, ipvt=None, tol=1e-14)

call (fcn, x, functkw, fjac=None)

defiter (fcn, x, iter, fnorm=None, functkw=None, quiet=0, iterstop=None, parinfo=None, format=None, pformat='%1.10g', dof=1)

enorm (vec)

call (fcn, x, functkw, fjac=None)

defiter (fcn, x, iter, fnorm=None, functkw=None, quiet=0, iterstop=None, parinfo=None, format=None, pformat='%1.10g', dof=1)

hyperspy.external.mpfit.mpfitexpr module

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

tie (p, ptied=None)

**hyperspy.external.mpfit.mpfitexpr**

<table>
<thead>
<tr>
<th><strong>mpfitexpr</strong></th>
<th><strong>func</strong>, <strong>x</strong>, <strong>y</strong>, <strong>err</strong>, <strong>start_params</strong>, **check=**True, **full_output=**False, **imports=**None, ****kw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fit the used defined expression to the data</td>
<td></td>
</tr>
<tr>
<td>Input: - <strong>func</strong>: string with the function definition - <strong>x</strong>: x vector - <strong>y</strong>: y vector - <strong>err</strong>: vector with the errors of y - <strong>start_params</strong>: the starting parameters for the fit</td>
<td></td>
</tr>
<tr>
<td>Output: - The tuple (<strong>params</strong>, <strong>yfit</strong>) with best-fit params and the values of <strong>func</strong> evaluated at <strong>x</strong></td>
<td></td>
</tr>
<tr>
<td>Keywords: - <strong>check</strong>: boolean parameter. If true(default) the function will be checked for sanity - <strong>full_output</strong>: boolean parameter. If True(default is False) then instead of best-fit parameters the mpfit object is returned - imports: list of strings, of optional modules to be imported, required to evaluate the function</td>
<td></td>
</tr>
<tr>
<td>Example: <strong>params</strong>, <strong>yfit</strong> = mpfitexpr('p[0]+p[2]*(x-p[1])', <strong>x</strong>, <strong>y</strong>, <strong>err</strong>, [0, 10, 1])</td>
<td></td>
</tr>
</tbody>
</table>

If you need to use numpy and scipy functions in your function, then you must use the full names of these functions, e.g.: numpy.sin, numpy.cos etc.

This function is motivated by mpfitexpr() from wonderful MPFIT IDL package written by Craig Markwardt

**hyperspy.external.mpfit.test_mpfit**

**Flin** (**x**, **p**)

**myfunctlin** (**p**, **fjac=**None, **x=**None, **y=**None, **err=**None)

**myfunctrosenbrock** (**p**, **fjac=**None)

3.1. hyperspy package
Module contents

Submodules

`hyperspy.external.progressbar module`

`hyperspy.external.progressbar.progressbar(*args, **kwargs)`

Uses `tqdm` progressbar. This function exists for wrapping purposes only. Original docstring follows:

Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

- **iterable** [iterable, optional] Iterable to decorate with a progressbar. Leave blank to manually manage the updates.
- **desc** [str, optional] Prefix for the progressbar.
- **total** [int, optional] The number of expected iterations. If unspecified, `len(iterable)` is used if possible. If float("inf") or as a last resort, only basic progress statistics are displayed (no ETA, no progressbar). If gui is True and this parameter needs subsequent updating, specify an initial arbitrary large positive integer, e.g. int(9e9).
- **leave** [bool, optional] If [default: True], keeps all traces of the progressbar upon termination of iteration.
- **file** [io.TextIOWrapper or io.StringIO, optional] Specifies where to output the progress messages (default: sys.stderr). Uses `file.write(str)` and `file.flush()` methods. For encoding, see `write_bytes`.
- **ncols** [int, optional] The width of the entire output message. If specified, dynamically resizes the progressbar to stay within this bound. If unspecified, attempts to use environment width. The fallback is a meter width of 10 and no limit for the counter and statistics. If 0, will not print any meter (only stats).
- **mininterval** [float, optional] Minimum progress display update interval [default: 0.1] seconds.
- **maxinterval** [float, optional] Maximum progress display update interval [default: 10] seconds. Automatically adjusts `miniters` to correspond to `mininterval` after long display update lag. Only works if `dynamic_miniters` or monitor thread is enabled.
- **miniters** [int, optional] Minimum progress display update interval, in iterations. If 0 and `dynamic_miniters`, will automatically adjust to equal `mininterval` (more CPU efficient, good for tight loops). If > 0, will skip display of specified number of iterations. Tweak this and `mininterval` to get very efficient loops. If your progress is erratic with both fast and slow iterations (network, skipping items, etc) you should set miniters=1.
- **ascii** [bool or str, optional] If unspecified or False, use unicode (smooth blocks) to fill the meter. The fallback is to use ASCII characters “ 123456789#”.
- **disable** [bool, optional] Whether to disable the entire progressbar wrapper [default: False]. If set to None, disable on non-TTY.
- **unit** [str, optional] String that will be used to define the unit of each iteration [default: it].
**unit_scale** [bool or int or float, optional] If 1 or True, the number of iterations will be reduced/scaled automatically and a metric prefix following the International System of Units standard will be added (kilo, mega, etc.) [default: False]. If any other non-zero number, will scale total and n.

**dynamic_ncols** [bool, optional] If set, constantly alters ncols to the environment (allowing for window resizes) [default: False].

**smoothing** [float, optional] Exponential moving average smoothing factor for speed estimates (ignored in GUI mode). Ranges from 0 (average speed) to 1 (current/instantaneous speed) [default: 0.3].

**bar_format** [str, optional] Specify a custom bar string formatting. May impact performance. [default: '{l_bar} {bar} {r_bar}'], where

- l_bar='{desc}: {percentage:3.0f}%|'
- r_bar='| {n_fmt}/ {total_fmt} [{elapsed}<{remaining}, '
- rate_fmt}{postfix}]

Possible vars: l_bar, bar, r_bar, n, n_fmt, total, total_fmt, percentage, rate, rate_fmt, rate_noinv, rate_noinv_fmt, rate_inv, rate_inv_fmt, elapsed, elapsed_s, remaining, remaining_s, desc, postfix, unit.

Note that a trailing “::” is automatically removed after { desc} if the latter is empty.

**initial** [int, optional] The initial counter value. Useful when restarting a progress bar [default: 0].

**position** [int, optional] Specify the line offset to print this bar (starting from 0) Automatic if unspecified. Useful to manage multiple bars at once (eg, from threads).

**postfix** [dict or *, optional] Specify additional stats to display at the end of the bar. Calls set_postfix(**postfix) if possible (dict).

**unit_divisor** [float, optional] [default: 1000], ignored unless unit_scale is True.

**write_bytes** [bool, optional] If (default: None) and file is unspecified, bytes will be written in Python 2. If True will also write bytes. In all other cases will default to unicode.

**gui** [bool, optional] WARNING: internal parameter - do not use. Use tqdm_gui(...) instead. If set, will attempt to use matplotlib animations for a graphical output [default: False].

**out** : decorated iterator.

### hyperspy.external.tifffile module

Read and write TIFF(r) files.

Tiffile is a package to (1) store numpy arrays in TIFF (Tagged Image File Format) files, and (2) read image and metadata from TIFF like files used in bioimaging.

Image and metadata can be read from TIFF, BigTIFF, OME-TIFF, STK, LSM, NIH, SGI, ImageJ, MicroManager, FluoView, ScanImage, SEQ, GEL, SVS, SCN, SIS, and GeoTIFF files.

Numpy arrays can be written to TIFF, BigTIFF, and ImageJ hyperstack compatible files in multi-page, memory-mappable, tiled, predicted, or compressed form.

Only a subset of the TIFF specification is supported, mainly uncompressed and losslessly compressed 1, 8, 16, 32 and 64-bit integer, 16, 32 and 64-bit float, grayscale and RGB(A) images. Specifically, reading slices of image data, image trees defined via SubIFDs, CCITT and OJPEG compression, chroma subsampling without JPEG compression, or IPTC and XMP metadata are not implemented.
TIFF(r), the Tagged Image File Format, is a trademark and under control of Adobe Systems Incorporated. BigTIFF allows for files greater than 4 GB. STK, LSM, FluoView, SGI, SEQ, GEL, and OME-TIFF, are custom extensions defined by Molecular Devices (Universal Imaging Corporation), Carl Zeiss MicroImaging, Olympus, Silicon Graphics International, Media Cybernetics, Molecular Dynamics, and the Open Microscopy Environment consortium respectively.

For command line usage run python -m tifffile --help

**Author**  Christoph Gohlke

**Organization**  Laboratory for Fluorescence Dynamics, University of California, Irvine

**Version**  2018.9.27

**Requirements**

- CPython 2.7 or 3.5+
- Numpy 1.14
- Imagecodecs 2018.9.22
- Matplotlib 2.2 (optional for plotting)
- Python 2 requires ‘futures’, ‘enum34’, and ‘pathlib’.

**Revisions**

2018.9.27  Read Olympus SIS (WIP). Allow to write non-BigTIFF files up to ~4 GB (bug fix). Fix parsing date and time fields in SEM metadata (bug fix). Detect some circular IFD references. Enable WebP codecs via imagecodecs. Add option to read TiffSequence from ZIP containers. Remove TiffFile.isnative. Move TIFF struct format constants out of TiffFile namespace.


2018.6.20  Save RGBA with unassociated extrasample by default (backward incompatible). Add option to specify ExtraSamples values.


2018.2.18  Pass 2293 tests. Always save RowsPerStrip and Resolution tags as required by TIFF standard. Do not use badly typed ImageDescription. Coerce bad ASCII string tags to bytes. Tuning of __str__ functions. Fix reading ‘undefined’ tag values (bug fix). Read and write ZSTD compressed data. Use hexdump to print byte strings. Determine TIFF byte order from data dtype in imsave. Add option to specify RowsPerStrip for compressed strips. Allow memory-map of arrays with non-native byte order. Attempt to handle ScanImage <=
5.1 files. Restore TiffPageSeries.pages sequence interface. Use numpy.frombuffer instead of fromstring to read from binary data. Parse GeoTIFF metadata. Add option to apply horizontal differencing before compression. Towards reading PerkinElmer QPTIFF (no test files). Do not index out of bounds data in tifffile.c unpackbits and decodelzw.


2017.5.23 Pass 1961 tests. Write correct number of SampleFormat values (bug fix). Use Adobe deflate code to write ZIP compressed files. Add option to pass tag values as packed binary data for writing. Defer tag validation to attribute access. Use property instead of lazyattr decorator for simple expressions.

2017.3.17 Write IFDs and tag values on word boundaries. Read ScanImage metadata. Remove is_rgb and is_indexed attributes from TiffFile. Create files used by doctests.

2017.1.12 Read Zeiss SEM metadata. Read OME-TIFF with invalid references to external files. Rewrite C LZW decoder (5x faster). Read corrupted LSM files missing EOI code in LZW stream.

2017.1.1 Add option to append images to existing TIFF files. Read files without pages. Read S-FEG and Helios NanoLab tags created by FEI software. Allow saving Color Filter Array (CFA) images. Add info functions returning more information about TiffFile and TiffPage. Add option to read specific pages only. Remove maxpages argument (backward incompatible). Remove test_tifffile function.


2016.6.21 Do not always memmap contiguous data in page series.

2016.5.13 Add option to specify resolution unit. Write grayscale images with extra samples when planarconfig is specified. Do not write RGB color images with 2 samples. Reorder TiffWriter.save keyword arguments (backward incompatible).

2016.4.18 Pass 1932 tests. TiffWriter, imread, and imsave accept open binary file streams.

2016.4.13 Correctly handle reversed fill order in 2 and 4 bps images (bug fix). Implement reverse_bitorder in C.

2016.03.18 Fix saving additional ImageJ metadata.

2016.2.22 Pass 1920 tests. Write 8 bytes double tag values using offset if necessary (bug fix). Add option to disable writing second image description tag. Detect tags with incorrect counts. Disable color mapping for LSM.

3.1. hyperspy package
2015.11.13 Read LSM 6 mosaics. Add option to specify directory of memory-mapped files. Add command line options to specify vmin and vmax values for colormapping.

2015.10.06 New helper function to apply colormaps. Renamed is_palette attributes to is_indexed (backward incompatible). Color-mapped samples are now contiguous (backward incompatible). Do not color-map ImageJ hyperstacks (backward incompatible). Towards reading Leica SCN.

2015.9.25 Read images with reversed bit order (FillOrder is LSB2MSB).

2015.9.21 Read RGB OME-TIFF. Warn about malformed OME-XML.


2014.8.24 TiffWriter class for incremental writing images. Simplify examples.

2014.8.19 Add memmap function to FileHandle. Add function to determine if image data in TiffPage is memory-mappable. Do not close files if multifile_close parameter is False.


2014.2.05 Save rational numbers as type 5 (bug fix).


2013.11.3 Allow zlib compress data in imsave function (optional). Memory-map contiguous image data (optional).


2012.10.18 Accept file like objects (read from OIB files).

2012.8.21 Rename TIFFfile to TiffFile and TIFFpage to TiffPage. TiffSequence class for reading sequence of TIFF files. Read UltraQuant tags. Allow float numbers as resolution in imsave function.

2012.8.3 Read MD GEL tags and NIH Image header.

2012.7.25 Read ImageJ tags. . . .

Notes

The API is not stable yet and might change between revisions.
Tested on little-endian platforms only.
Python 2.7 and 3.4 are deprecated.
Other libraries for reading scientific TIFF files from Python:

- Python-bioformats
- Imread
- GDAL
- OpenSlide-python
- PyLibTiff
- SimpleITK
- PyLSM
- PyMca.TiffIO.py (same as fabio.TiffIO)
- BioImageXD.Readers
- Cellcognition.io
- pymimage
- pytiff

Acknowledgements

- Egor Zindy, University of Manchester, for lsm_scan_info specifics.
- Wim Lewis for a bug fix and some LSM functions.
- Hadrien Mary for help on reading MicroManager files.
- Christian Kliche for help writing tiled and color-mapped files.

References


2) TIFF File Format FAQ. http://www.awaresystems.be/imaging/tiff/faq.html


Examples

Save a 3D numpy array to a multi-page, 16-bit grayscale TIFF file:

```python
>>> data = numpy.random.randint(0, 2**16, (4, 301, 219), 'uint16')
>>> imsave('temp.tif', data, photometric='minisblack')
```

Read the whole image stack from the TIFF file as numpy array:

```python
>>> image_stack = imread('temp.tif')
>>> image_stack.shape
(4, 301, 219)
>>> image_stack.dtype
dtype('uint16')
```

Read the image from first page (IFD) in the TIFF file:

```python
>>> image = imread('temp.tif', key=0)
>>> image.shape
(301, 219)
```

Read images from a sequence of TIFF files as numpy array:

```python
>>> image_sequence = imread(['temp.tif', 'temp.tif'])
>>> image_sequence.shape
(2, 4, 301, 219)
```

Save a numpy array to a single-page RGB TIFF file:

```python
>>> data = numpy.random.randint(0, 255, (256, 256, 3), 'uint8')
>>> imsave('temp.tif', data, photometric='rgb')
```

Save a floating-point array and metadata, using zlib compression:

```python
>>> data = numpy.random.rand(2, 5, 3, 301, 219).astype('float32')
>>> imsave('temp.tif', data, compress=6, metadata={'axes': 'TZCYX'})
```

Save a volume with xyz voxel size 2.6755x2.6755x3.9474 µm³ to ImageJ file:

```python
>>> volume = numpy.random.randn(57*256*256).astype('float32')
>>> volume.shape = 1, 57, 1, 256, 256, 1   # dimensions in TZCYXS order
>>> imsave('temp.tif', volume, imagej=True, resolution=(0.373759, 0.373759),
...        metadata={'spacing': 3.947368, 'unit': 'um'})
```

Read hyperstack and metadata from ImageJ file:

```python
>>> with TiffFile('temp.tif') as tif:
...     imagej_hyperstack = tif.asarray()
...     imagej_metadata = tif.imagej_metadata
>>> imagej_hyperstack.shape
(57, 256, 256)
>>> imagej_metadata['slices']
57
```

Create an empty TIFF file and write to the memory-mapped numpy array:
Memory-map image data in the TIFF file:

```python
>>> memmap_image = memmap('temp.tif', shape=(256, 256), dtype='float32')
>>> memmap_image[255, 255] = 1.0
>>> memmap_image.flush()
>>> memmap_image.shape, memmap_image.dtype
((256, 256), dtype('float32'))
>>> del memmap_image
```

Successively append images to a BigTIFF file:

```python
>>> data = numpy.random.randint(0, 255, (5, 2, 3, 301, 219), 'uint8')
>>> with TiffWriter('temp.tif', bigtiff=True) as tif:
...     for i in range(data.shape[0]):
...         tif.save(data[i], compress=6, photometric='minisblack')
```

Iterate over pages and tags in the TIFF file and successively read images:

```python
>>> with TiffFile('temp.tif') as tif:
...     image_stack = tif.asarray()
...     for page in tif.pages:
...         for tag in page.tags.values():
...             tag_name, tag_value = tag.name, tag.value
...             image = page.asarray()
```

Save two image series to a TIFF file:

```python
>>> data0 = numpy.random.randint(0, 255, (301, 219, 3), 'uint8')
>>> data1 = numpy.random.randint(0, 255, (5, 301, 219), 'uint16')
>>> with TiffWriter('temp.tif') as tif:
...     tif.save(data0, compress=6, photometric='rgb')
...     tif.save(data1, compress=6, photometric='minisblack')
```

Read the second image series from the TIFF file:

```python
>>> series1 = imread('temp.tif', series=1)
>>> series1.shape
(5, 301, 219)
```

Read a image stack from a sequence of TIFF files with a file name pattern:

```python
>>> imssave('temp_C001T001.tif', numpy.random.rand(64, 64))
>>> imssave('temp_C001T002.tif', numpy.random.rand(64, 64))
>>> image_sequence = TiffSequence('temp_C001*.tif')
>>> image_sequence.shape
(1, 2)
>>> image_sequence.axes
'CT'
>>> data = image_sequence.asarray()
>>> data.shape
(1, 2, 64, 64)
```
hyperspy.external.tifffile.imsave(file, data=None, shape=None, dtype=None, bigsize=4261412864, **kwargs)

Write numpy array to TIFF file.

Refer to the TiffWriter class and member functions for documentation.

Parameters

- **file** *(str or binary stream)* – File name or writable binary stream, such as an open file or BytesIO.
- **data** *(array_like)* – Input image. The last dimensions are assumed to be image depth, height, width, and samples. If None, an empty array of the specified shape and dtype is saved to file. Unless ‘byteorder’ is specified in ‘kwargs’, the TIFF file byte order is determined from the data’s dtype or the dtype argument.
- **shape** *(tuple)* – If ‘data’ is None, shape of an empty array to save to the file.
- **dtype** *(numpy.dtype)* – If ‘data’ is None, data-type of an empty array to save to the file.
- **bigsize** *(int)* – Create a BigTIFF file if the size of data in bytes is larger than this threshold and ‘imagej’ or ‘truncate’ are not enabled. By default, the threshold is 4 GB minus 32 MB reserved for metadata. Use the ‘bigtiff’ parameter to explicitly specify the type of file created.
- **kwargs** *(dict)* – Parameters ‘append’, ‘byteorder’, ‘bigtiff’, and ‘imagej’, are passed to TiffWriter(). Other parameters are passed to TiffWriter.save().

Returns *offset*, *bytecount* – If the image data are written contiguously, return offset and bytecount of image data in the file.

Return type  tuple or None

hyperspy.external.tifffile.imread(files, **kwargs)

Return image data from TIFF file(s) as numpy array.

Refer to the TiffFile class and member functions for documentation.

Parameters

- **files** *(str, binary stream, or sequence)* – File name, seekable binary stream, glob pattern, or sequence of file names.
- **kwargs** *(dict)* – Parameters ‘multifile’ and ‘is_ome’ are passed to the TiffFile class. The ‘pattern’ parameter is passed to the TiffSequence class. Other parameters are passed to the asarray functions. The first image series is returned if no arguments are provided.

hyperspy.external.tifffile.imshow(data, title=None, vmin=0, vmax=None, cmap=None, bitspersample=None, photometric='RGB', interpolation=None, dpi=96, figure=None, subplot=111, maxdim=32768, **kwargs)

Plot n-dimensional images using matplotlib.pyplot.

Return figure, subplot and plot axis. Requires pyplot already imported C{from matplotlib import pyplot}.

Parameters

- **bitspersample** *(int or None)* – Number of bits per channel in integer RGB images.
- **photometric** *(str, {'MINISWHITE', 'MINISBLACK', 'RGB', 'PALETTE'})* – The color space of the image data.
- **title** *(str)* – Window and subplot title.
- **figure**(matplotlib.figure.Figure (optional)) – Matplotlib to use for plotting.
- **subplot**(int) – A matplotlib.pyplot.subplot axis.
- **maxdim**(int) – maximum image width and length.
- **kwargs**(optional) – Arguments for matplotlib.pyplot.imshow.

```python
hyperspy.external.tifffile.memmap(filename, shape=None, dtype=None, page=None, series=0, mode='r+', **kwargs)
```

Return memory-mapped numpy array stored in TIFF file.

Memory-mapping requires data stored in native byte order, without tiling, compression, predictors, etc. If ‘shape’ and ‘dtype’ are provided, existing files will be overwritten or appended to depending on the ‘append’ parameter. Otherwise the image data of a specified page or series in an existing file will be memory-mapped. By default, the image data of the first page series is memory-mapped. Call flush() to write any changes in the array to the file. Raise ValueError if the image data in the file is not memory-mappable.

### Parameters
- **filename**(str) – Name of the TIFF file which stores the array.
- **shape**(tuple) – Shape of the empty array.
- **dtype**(numpy.dtype) – Data-type of the empty array.
- **page**(int) – Index of the page which image data to memory-map.
- **series**(int) – Index of the page series which image data to memory-map.
- **mode**({'r+', 'r', 'c'}, optional) – The file open mode. Default is to open existing file for reading and writing (‘r+’).
- **kwargs**(dict) – Additional parameters passed to imsave() or TiffFile().

```python
class hyperspy.external.tifffile.TiffFile(arg, name=None, offset=None, size=None, multifile=True, movie=None, **kwargs)
```

Initialize instance from file.

TiffFile instances must be closed using the ‘close’ method, which is automatically called when using the ‘with’ context manager.

- **pages**
  - Type TiffPages

- **series**
  - Type list of TiffPageSeries

- **is_flag**
  - Type bool
  - If True, file is of a certain format. Flags are: bigtiff, movie, shaped, ome, imagej, stk, lsm, fluoview, nih, vista, micromanager, metaseries, mdgel, mediacy, tvips, fei, sem, scn, svs, scanimage, andor, epics, ndpi, pilatus, aptiff.

All attributes are read-only.
Parameters

- `arg (str or open file)` – Name of file or open file object. The file objects are closed in TiffFile.close().

- `name (str)` – Optional name of file in case ‘arg’ is a file handle.

- `offset (int)` – Optional start position of embedded file. By default, this is the current file position.

- `size (int)` – Optional size of embedded file. By default, this is the number of bytes from the ‘offset’ to the end of the file.

- `multifile (bool)` – If True (default), series may include pages from multiple files. Currently applies to OME-TIFF only.

- `movie (bool)` – If True, assume that later pages differ from first page only by data offsets and byte counts. Significantly increases speed and reduces memory usage when reading movies with thousands of pages. Enabling this for non-movie files will result in data corruption or crashes. Python 3 only.

- `kwargs (bool)` – ‘is_ome’: If False, disable processing of OME-XML metadata.

`andor_metadata`  
Attribute whose value is computed on first access.

`asarray (key=None, series=None, out=None, validate=True, maxworkers=None)`  
Return image data from multiple TIFF pages as numpy array.

By default, the data from the first series is returned.

Parameters

- `key (int, slice, or sequence of page indices)` – Defines which pages to return as array.

- `series (int or TiffPageSeries)` – Defines which series of pages to return as array.

- `out (numpy.ndarray, str, or file-like object; optional)` – Buffer where image data will be saved. If None (default), a new array will be created. If numpy.ndarray, a writable array of compatible dtype and shape. If ‘memmap’, directly memory-map the image data in the TIFF file if possible; else create a memory-mapped array in a temporary file. If str or open file, the file name or file object used to create a memory-map to an array stored in a binary file on disk.

- `validate (bool)` – If True (default), validate various tags. Passed to TiffPage.asarray().

- `maxworkers (int or None)` – Maximum number of threads to concurrently get data from pages or tiles. If None (default), mutli-threading is enabled if data are compressed. If 0, up to half the CPU cores are used. If 1, mutli-threading is disabled. Reading data from file is limited to a single thread. Using multiple threads can significantly speed up this function if the bottleneck is decoding compressed data, e.g. in case of large LZW compressed LSM files or JPEG compressed tiled slides. If the bottleneck is I/O or pure Python code, using multiple threads might be detrimental.

Returns Image data from the specified pages. See the TiffPage.asarray function for what kind of operations are applied (or not) to the raw data stored in the file.

Return type `numpy.ndarray`

`property byteorder`
close()
    Close open file handle(s).

epics_metadata
    Attribute whose value is computed on first access.

fei_metadata
    Attribute whose value is computed on first access.

property filehandle
    Return file handle.

property filename
    Return name of file handle.

flags
    Attribute whose value is computed on first access.

fluoview_metadata
    Attribute whose value is computed on first access.

fstat
    Attribute whose value is computed on first access.

property geotiff_metadata
    Return GeoTIFF metadata from first page as dict.

imagej_metadata
    Attribute whose value is computed on first access.

property is_bigtiff

is_mdgel
    Attribute whose value is computed on first access.

property is_movie
    Return if file is a movie.

lsm_metadata
    Attribute whose value is computed on first access.

mdgel_metadata
    Attribute whose value is computed on first access.

metaseries_metadata
    Attribute whose value is computed on first access.

micromanager_metadata
    Attribute whose value is computed on first access.

nih_metadata
    Attribute whose value is computed on first access.

ome_metadata
    Attribute whose value is computed on first access.

pilatus_metadata
    Attribute whose value is computed on first access.

qptiff_metadata
    Attribute whose value is computed on first access.

scanimage_metadata
    Attribute whose value is computed on first access.
sem_metadata
Attribute whose value is computed on first access.

series
Attribute whose value is computed on first access.

shaped_metadata
Attribute whose value is computed on first access.

sis_metadata
Attribute whose value is computed on first access.

stk_metadata
Attribute whose value is computed on first access.

tvips_metadata
Attribute whose value is computed on first access.

class hyperspy.external.tifffile.TiffWriter(file, bigtiff=False, byteorder=None, append=False, imagej=False)

Bases: object

Write numpy arrays to TIFF file.

TiffWriter instances must be closed using the ‘close’ method, which is automatically called when using the ‘with’ context manager.

TiffWriter’s main purpose is saving nD numpy array’s as TIFF, not to create any possible TIFF format. Specifically, JPEG compression, SubIFDs, ExifIFD, or GPSIFD tags are not supported.

Open a TIFF file for writing.

An empty TIFF file is created if the file does not exist, else the file is overwritten with an empty TIFF file unless ‘append’ is true. Use bigtiff=True when creating files larger than 4 GB.

Parameters

• file (str, binary stream, or FileHandle) – File name or writable binary stream, such as an open file or BytesIO.

• bigtiff (bool) – If True, the BigTIFF format is used.

• byteorder ({'<', '>', '='}) – The endianness of the data in the file. By default, this is the system’s native byte order.

• append (bool) – If True and ‘file’ is an existing standard TIFF file, image data and tags are appended to the file. Appending data may corrupt specifically formatted TIFF files such as LSM, STK, ImageJ, NIH, or FluoView.

• imagej (bool) – If True, write an ImageJ hyperstack compatible file. This format can handle data types uint8, uint16, or float32 and data shapes up to 6 dimensions in TZCYXS order. RGB images (S=3 or S=4) must be uint8. ImageJ’s default byte order is big-endian but this implementation uses the system’s native byte order by default. ImageJ does not support BigTIFF format or non DEFLATE compression. The ImageJ file format is undocumented.

close()

Write remaining pages and close file handle.

save(data=None, shape=None, dtype=None, returnoffset=False, photometric=None, planarconfig=None, extrasamples=None, file=None, contiguous=True, align=16, truncate=False, compress=0, rowsperstrip=None, predictor=False, colormap=None, description=None, date-time=None, resolution=None, subfiletype=0, software='tifffile.py', metadata={}, ijmetadata=None, extratags=())
Write numpy array and tags to TIFF file.

The data shape’s last dimensions are assumed to be image depth, height (length), width, and samples. If a colormap is provided, the data’s dtype must be uint8 or uint16 and the data values are indices into the last dimension of the colormap. If ‘shape’ and ‘dtype’ are specified, an empty array is saved. This option cannot be used with compression or multiple tiles. Image data are written uncompressed in one strip per plane by default. Dimensions larger than 2 to 4 (depending on photometric mode, planar configuration, and SGI mode) are flattened and saved as separate pages. The SampleFormat and BitsPerSample tags are derived from the data type.

**Parameters**

- **data** *(numpy.ndarray or None)* – Input image array.
- **shape** *(tuple or None)* – Shape of the empty array to save. Used only if ‘data’ is None.
- **dtype** *(numpy.dtype or None)* – Data-type of the empty array to save. Used only if ‘data’ is None.
- **returnoffset** *(bool)* – If True and the image data in the file is memory-mappable, return the offset and number of bytes of the image data in the file.
- **photometric** *({'MINISBLACK', 'MINISWHITE', 'RGB', 'PALETTE', 'CFA'})* – The color space of the image data. By default, this setting is inferred from the data shape and the value of colormap. For CFA images, DNG tags must be specified in ‘extratags’.
- **planarconfig** *({'CONTIG', 'SEPARATE'})* – Specifies if samples are stored contiguous or in separate planes. By default, this setting is inferred from the data shape. If this parameter is set, extra samples are used to store grayscale images. ‘CONTIG’: last dimension contains samples. ‘SEPARATE’: third last dimension contains samples.
- **extrasamples** *(tuple of {'UNSPECIFIED', 'ASSOCALPHA', 'UNASSALPHA'})* – Defines the interpretation of extra components in pixels. ‘UNSPECIFIED’: no transparency information (default). ‘ASSOCALPHA’: single, true transparency with pre-multiplied color. ‘UNASSALPHA’: independent transparency masks.
- **tile** *(tuple of int)* – The shape (depth, length, width) of image tiles to write. If None (default), image data are written in strips. The tile length and width must be a multiple of 16. If the tile depth is provided, the SGI ImageDepth and TileDepth tags are used to save volume data. Unless a single tile is used, tiles cannot be used to write contiguous files. Few software can read the SGI format, e.g. MeVisLab.
- **contiguous** *(bool)* – If True (default) and the data and parameters are compatible with previous ones, if any, the image data are stored contiguous after the previous one. Parameters ‘photometric’ and ‘planarconfig’ are ignored. Parameters ‘description’, date-time’, and ‘extratags’ are written to the first page of a contiguous series only.
- **align** *(int)* – Byte boundary on which to align the image data in the file. Default 16. Use mmap.ALLOCATIONGRANULARITY for memory-mapped data. Following contiguous writes are not aligned.
- **truncate** *(bool)* – If True, only write the first page including shape metadata if possible (uncompressed, contiguous, not tiled). Other TIFF readers will only be able to read part of the data.
- **compress** *(int or str or (str, int))* – If 0 (default), data are written uncompressed. If 0-9, the level of ADOBE_DEFLATE compression. If a str, one of TIFF.COMPRESSION, e.g. ‘LZMA’ or ‘ZSTD’. If a tuple, first item is one of
TIFF.COMPRESSION and second item is compression level. Compression cannot be
used to write contiguous files.

- **rowsperstrip (int)** – The number of rows per strip used for compression. Uncom-
presed data are written in one strip per plane.

- **predictor (bool)** – If True, apply horizontal differencing or floating point predictor
before compression.

- **colormap (numpy.ndarray)** – RGB color values for the corresponding data value.
Must be of shape (3, 2**(data.itemsize*8)) and dtype uint16.

- **description (str)** – The subject of the image. Must be 7-bit ASCII. Cannot be used
with the ImageJ format. Saved with the first page only.

- **datetime (datetime)** – Date and time of image creation in ‘%Y:%m:%d
%H:%M:%S’ format. If None (default), the current date and time is used. Saved with
the first page only.

- **resolution (float, float, str) or (int, int), (int, int)[, str])** – X and Y resolutions in pixels per resolution unit as float or
rational numbers. A third, optional parameter specifies the resolution unit, which must be
None (default for ImageJ), ‘INCH’ (default), or ‘CENTIMETER’.

- **subfiletype (int)** – Bitfield to indicate the kind of data. Set bit 0 if the image is a
reduced-resolution version of another image. Set bit 1 if the image is part of a multi-page
image. Set bit 2 if the image is transparency mask for another image (photometric must be
MASK, SamplesPerPixel and BitsPerSample must be 1).

- **software (str)** – Name of the software used to create the file. Must be 7-bit ASCII.
Saved with the first page only.

- **metadata (dict)** – Additional meta data to be saved along with shape information in
JSON or ImageJ formats in an ImageDescription tag. If None, do not write a second
ImageDescription tag. Strings must be 7-bit ASCII. Saved with the first page only.

- **ijmetadata (dict)** – Additional meta data to be saved in application specific IJMeta-
data and IJMetadataByteCounts tags. Refer to the imagej_metadata_tag function for valid
keys and values. Saved with the first page only.

- **extratags (sequence of tuples)** – Additional tags as [code, dtype, count,
value, writeonce].

  - **code** [int] The TIFF tag Id.
  - **dtype** [str] Data type of items in ‘value’ in Python struct format. One of B, s, H, I, 2I, b,
h, i, 2i, f, d, Q, or q.
  - **count** [int] Number of data values. Not used for string or byte string values.
  - **value** [sequence] ’Count’ values compatible with ‘dtype’. Byte strings must contain count
values of dtype packed as binary data.
  - **writeonce** [bool] If True, the tag is written to the first page only.

**class hyperspy.external.tifffile.TiffSequence** (files=None, imread=<class 'hyper-
spy.external.tifffile.TiffFile'>, pattern=‘axes’, container=None, sort=None,
*args, **kwargs)

Bases: object

Sequence of TIFF files.

The image data in all files must match shape, dtype, etc.
files
List of file names.
  Type list

shape
Shape of image sequence. Excludes shape of image array.
  Type tuple

axes
Labels of axes in shape.
  Type str

Initialize instance from multiple files.

Parameters
- **files** (*str, pathlib.Path, or sequence thereof*) – Glob filename pattern or sequence of file names. Default is ‘*.tif’. Binary streams are not supported.
- **imread** (*function or class*) – Image read function or class with asarray function returning numpy array from single file.
- **pattern** (*str*) – Regular expression pattern that matches axes names and sequence indices in file names. By default, the pattern matches Olympus OIF and Leica TIFF series.
- **container** (*str, container instance, or None*) – Name or open instance of ZIP file in which files are stored.
- **sort** (*function or None*) – Sort function used to sort file names when ‘files’ is a pattern. The default (None) is natural_sorted.

**exception ParseError**
Bases: Exception

**asarray** (*file=None, out=None, *args, **kwargs*)
Read image data from files and return as numpy array.
  The args and kwargs parameters are passed to the imread function.
  Raise IndexError or ValueError if image shapes do not match.

**close** ()

**class hyperspy.external.tifffile.FileHandle** (*file, mode='rb', name=None, offset=None, size=None*)
Bases: object

Binary file handle.
A limited, special purpose file handler that can:
- handle embedded files (for CZI within CZI files)
- re-open closed files (for multi-file formats, such as OME-TIFF)
- read and write numpy arrays and records from file like objects

Only ‘rb’ and ‘wb’ modes are supported. Concurrently reading and writing of the same stream is untested.

When initialized from another file handle, do not use it unless this FileHandle is closed.

**name**
Name of the file.
  Type str
path
    Absolute path to file.
    Type  str
size
    Size of file in bytes.
    Type  int
is_file
    If True, file has a filno and can be memory-mapped.
    Type  bool

All attributes are read-only.

Initialize file handle from file name or another file handle.

Parameters

- **file** *(str, pathlib.Path, binary stream, or FileHandle)* – File name or seekable binary stream, such as an open file or BytesIO.
- **mode** *(str)* – File open mode in case ‘file’ is a file name. Must be ‘rb’ or ‘wb’.
- **name** *(str)* – Optional name of file in case ‘file’ is a binary stream.
- **offset** *(int)* – Optional start position of embedded file. By default, this is the current file position.
- **size** *(int)* – Optional size of embedded file. By default, this is the number of bytes from the ‘offset’ to the end of the file.

**close()**
    Close file.

**property closed**

**property dirname**

**flush()**
    Flush write buffers if applicable.

**is_file**

**property lock**

**memmap_array** *(dtype, shape, offset=0, mode='r', order='C')*
    Return numpy.memmap of data stored in file.

**property name**

**open()**
    Open or re-open file.

**property path**

**read(size=-1)**
    Read ‘size’ bytes from file, or until EOF is reached.

**read_array** *(dtype, count=-1, sep=',', chunksize=33554432, out=None, native=False)*
    Return numpy array from file.
    Work around numpy issue #2230, “numpy.fromfile does not accept StringIO object” https://github.com/numpy/numpy/issues/2230.
read_record(dtype, shape=1, byteorder=None)
Return numpy record from file.

seek(offset, whence=0)
Set file’s current position.

property size

tell()
Return file’s current position.

write(bytestring)
Write bytestring to file.

write_array(data)
Write numpy array to binary file.

write_empty(size)
Append size bytes to file. Position must be at end of file.

class hyperspy.external.tifffile.TiffPage(parent, index, keyframe=None)
Bases: object
TIFF image file directory (IFD).

index
Index of page in file.
  Type int
dtype
Data type (native byte order) of the image in IFD.
  Type numpy.dtype or None
shape
Dimensions of the image in IFD.
  Type tuple
axes
  Type str
tags
Dictionary of tags in IFD. {tag.name: TiffTag}
  Type dict
colormap
Color look up table, if exists.
  Type numpy.ndarray

All attributes are read-only.

Notes
The internal, normalized ‘_shape’ attribute is 6 dimensional:

Initialize instance from file.

The file handle position must be at offset to a valid IFD.

**andor_tags**

Attribute whose value is computed on first access.

**asarray** *(out=None, squeeze=True, lock=None, reopen=True, maxsize=17592186044416, maxworkers=None, validate=True)*

Read image data from file and return as numpy array.

Raise ValueError if format is unsupported.

**Parameters**

- **out** (numpy.ndarray, str, or file-like object; optional) – Buffer where image data will be saved. If None (default), a new array will be created. If numpy.ndarray, a writable array of compatible dtype and shape. If ‘memmap’, directly memory-map the image data in the TIFF file if possible; else create a memory-mapped array in a temporary file. If str or open file, the file name or file object used to create a memory-map to an array stored in a binary file on disk.

- **squeeze** (bool) – If True, all length-1 dimensions (except X and Y) are squeezed out from the array. If False, the shape of the returned array might be different from the page.shape.

- **lock** ((RLock, NullContext)) – A reentrant lock used to synchronize reads from file. If None (default), the lock of the parent’s filehandle is used.

- **reopen** (bool) – If True (default) and the parent file handle is closed, the file is temporarily re-opened and closed if no exception occurs.

- **maxsize** (int or None) – Maximum size of data before a ValueError is raised. Can be used to catch DOS. Default: 16 TB.

- **maxworkers** (int or None) – Maximum number of threads to concurrently decode tile data. If None (default), up to half the CPU cores are used for compressed tiles. See remarks in TiffFile.asarray.

- **validate** (bool) – If True (default), validate various parameters. If None, only validate parameters and return None.

**Returns** Numpy array of decompressed, depredicted, and unpacked image data read from Strip/Tile Offsets/ByteCounts, formatted according to shape and dtype metadata found in tags and parameters. Photometric conversion, pre-multiplied alpha, orientation, and colorimetry corrections are not applied. Specifically, CMYK images are not converted to RGB, MinIsWhite images are not inverted, and color palettes are not applied.

**Return type** numpy.ndarray

**aspage()**

**asrgb** *(uint8=False, alpha=None, colormap=None, dmin=None, dmax=None, *args, **kwargs)*

Return image data as RGB(A).

Work in progress.

**bitpersample = 1**

**colormap = None**

**compression = 1**
description = ''
description1 = ''

epics_tags
    Attribute whose value is computed on first access.

extrasamples = 1
fillorder = 1

flags
    Attribute whose value is computed on first access.

g6eiff_tags
    Attribute whose value is computed on first access.

imagedepth = 1
image_length = 0
image_width = 0

property is_andor
    Page contains Andor Technology tags.

property is_chroma_subsampled
    Page contains chroma subsampled image.

is_contiguous
    Attribute whose value is computed on first access.

property is_epics
    Page contains EPICS areaDetector tags.

property is_fei
    Page contains SFEG or HELIOS metadata.

is_final
    Attribute whose value is computed on first access.

property is_fluoview
    Page contains FluoView MM_STAMP tag.

property is_geotiff
    Page contains GeoTIFF metadata.

is_imagej
    Attribute whose value is computed on first access.

property is_lsm
    Page contains CZ_LSMINFO tag.

property is_mask
    Page is transparency mask for another image.

property is_mdgel
    Page contains MDFileTag tag.

property is_mediacy
    Page contains Media Cybernetics Id tag.

is_memmappable
    Attribute whose value is computed on first access.
property is_metaseries
Page contains MDS MetaSeries metadata in ImageDescription tag.

property is_micromanager
Page contains Micro-Manager metadata.

property is_mrc
Page is part of Mixed Raster Content.

property is_multipage
Page is part of multi-page image.

is_ndpi
Attribute whose value is computed on first access.

property is_nih
Page contains NIH image header.

property is_ome
Page contains OME-XML in ImageDescription tag.

property is_pilatus
Page contains Pilatus tags.

property is_qptiff
Page contains PerkinElmer tissue images metadata.

property is_reduced
Page is reduced image of another image.

property is_scanimage
Page contains ScanImage metadata.

property is_scn
Page contains Leica SCN XML in ImageDescription tag.

property is_sem
Page contains Zeiss SEM metadata.

property is_sgi
Page contains SGI image and tile depth tags.

is_shaped
Attribute whose value is computed on first access.

property is_sis
Page contains Olympus SIS metadata.

property is_stk
Page contains UIC2Tag tag.

property is_svs
Page contains Aperio metadata.

property is_tiled
Page contains tiled image.

property is_tvips
Page contains TVIPS metadata.

property is_vista
Software tag is ‘ISS Vista’.

property keyframe
Returns number of array dimensions.

Attribute whose value is computed on first access.

Attribute whose value is computed on first access.

Returns number of elements in array.

Lightweight TIFF image file directory (IFD).

Only a limited number of tag values are read from file, e.g. StripOffsets, and StripByteCounts. Other tag values are assumed to be identical with a specified TiffPage instance, the keyframe.

TiffFrame is intended to reduce resource usage and speed up reading data from file, not for introspection of metadata.

Not compatible with Python 2.

Read specified tags from file.

The file handle position must be at the offset to a valid IFD.

Read image data from file and return as numpy array.

Return TiffPage from file.

Read image data from file and return RGB image as numpy array.

Return offset and size of contiguous data, else None.
is_mdgel = False

property is_memmappable
    Return if page’s image data in file can be memory-mapped.

keyframe

offset

property offsets_bytecounts
    Return simplified offsets and bytecounts.

parent

tags = {}

class hyperspy.external.tifffile.TiffTag (parent, tagheader, tagoffset)
    Bases: object
    TIFF tag structure.

    name
        Name of tag.
        Type string

    code
        Decimal code of tag.
        Type int

    dtype
        Datatype of tag data. One of TIFF DATA_FORMATS.
        Type str

    count
        Number of values.
        Type int

    value
        Tag data as Python object.
        Type various types

    ImageSourceData
        Location of value in file.
        Type int

    All attributes are read-only.

Initialize instance from tag header.

exception Error
    Bases: Exception

    code
    count
dtype
    property name
    value
valueoffset

class hyperspy.external.tifffile.lazyattr(func)
    Bases: object
    
    Attribute whose value is computed on first access.

    func

class hyperspy.external.tifffile.natural_sorted(iterable)
    Return human sorted list of strings.
    E.g. for sorting file names.

    >>> natural_sorted(['f1', 'f2', 'f10'])
    ['f1', 'f2', 'f10']

    hyperspy.external.tifffile.stripnull(string, null=b'\x00')
    Return string truncated at first null character.
    Clean NULL terminated C strings. For unicode strings use null='0'.

    >>> stripnull(b'string\x00')
    b'string'
    >>> stripnull('string\x00', null='\0')
    'string'

    hyperspy.external.tifffile.transpose_axes(image, axes, asaxes='CTZYX')
    Return image with its axes permuted to match specified axes.
    A view is returned if possible.

    >>> transpose_axes(numpy.zeros((2, 3, 4, 5)), 'TYXC', asaxes='CTZYX').shape
    (5, 2, 1, 3, 4)

    hyperspy.external.tifffile.squeeze_axes(shape, axes, skip='XY')
    Return shape and axes with single-dimensional entries removed.
    Remove unused dimensions unless their axes are listed in 'skip'.

    >>> squeeze_axes((5, 1, 2, 1, 1), 'TZYXC')
    ((5, 2, 1), 'TYX')

    hyperspy.external.tifffile.create_output(out, shape, dtype, mode='w+', suffix='.memmap')
    Return numpy array where image data of shape and dtype can be copied.
    The ‘out’ parameter may have the following values or types:

    None  An empty array of shape and dtype is created and returned.
    numpy.ndarray  An existing writable array of compatible dtype and shape. A view of the same array is returned after verification.
    ‘memmap’ or ‘memmap:tempdir’ A memory-map to an array stored in a temporary binary file on disk is created and returned.
    str or open file  The file name or file object used to create a memory-map to an array stored in a binary file on disk. The created memory-mapped array is returned.

    hyperspy.external.tifffile.repeat_nd(a, repeats)
    Return read-only view into input array with elements repeated.
HyperSpy Documentation, Release 1.5.1.dev

Zoom nD image by integer factors using nearest neighbor interpolation (box filter).
Parameters
• a (array_like) – Input array.
• repeats (sequence of int) – The number of repetitions to apply along each dimension of input array.
Example
>>> repeat_nd([[1, 2], [3, 4]], (2, 2))
array([[1, 1, 2, 2],
[1, 1, 2, 2],
[3, 3, 4, 4],
[3, 3, 4, 4]])

hyperspy.external.tifffile.format_size(size, threshold=1536)
Return file size as string from byte size.
>>> format_size(1234)
'1234 B'
>>> format_size(12345678901)
'11.50 GiB'

hyperspy.external.tifffile.product(iterable)
Return product of sequence of numbers.
Equivalent of functools.reduce(operator.mul, iterable, 1). Multiplying numpy integers might overflow.
>>> product([2**8, 2**30])
274877906944
>>> product([])
1

hyperspy.external.tifffile.xml2dict(xml, sanitize=True, prefix=None)
Return XML as dict.
>>> xml2dict('<?xml version="1.0" ?><root attr="name"><key>1</key></root>')
{'root': {'key': 1, 'attr': 'name'}}

hyperspy.external.tifffile.pformat(arg, width=79, height=24, compact=True)
Return pretty formatted representation of object as string.
Whitespace might be altered.
hyperspy.external.tifffile.str2bytes(s, encoding=’cp1252’)
Return bytes from unicode string.
Module contents
hyperspy.io_plugins package
Submodules

338

Chapter 3. hyperspy


hyperspy.io_plugins.blockfile module

hyperspy.io_plugins.blockfile.file_reader (filename, endianess='<', mmap_mode=None, lazy=False, **kwds)

hyperspy.io_plugins.blockfile.file_writer (filename, signal, **kwds)

hyperspy.io_plugins.blockfile.get_default_header (endianess='<')

Returns a header pre-populated with default values.

hyperspy.io_plugins.blockfile.get_header_dtype_list (endianess='<')

hyperspy.io_plugins.blockfile.get_header_from_signal (signal, endianess='<')

hyperspy.io_plugins.bruker module

class hyperspy.io_plugins.bruker.BCF_reader (filename, instrument=None)
    Bases: hyperspy.io_plugins.bruker.SFS_reader

Class to read bcf (Bruker hypermapping) file.

Inherits SFS_reader and all its attributes and methods.

Attributes: filename

Methods: check_index_valid, parse_hypermap

The class instantiates HyperHeader class as self.header attribute where all metadata, sum eds spectras, (SEM) images are stored.

add_filename_to_general (item)
    hypy helper method

check_index_valid (index)
    check and return if index is valid

parse_hypermap (index=None, downsample=1, cutoff_at_kV=None, lazy=False)
    Unpack the Delphi/Bruker binary spectral map and return numpy array in memory efficient way.

    Pure python/numpy implementation – slow, or cython/memoryview/numpy implimentation if compiled and present (fast) is used.

    Arguments: index – the index of hypermap in bcf if there is more than one
                hyper map in file.

    downsample – downsampling factor (integer). Differently than block_reduce from skimage.measure, 
                 the parser populates reduced array by suming results of pixels, thus having lower memory require- 
                 ments. (default 1)

    cutoff_at_kV – value in keV to truncate the array at. Helps reducing size of array. (default None)

    lazy – return dask.array (True) or numpy.array (False) (default False)

    Returns: numpy or dask array of bruker hypermap, with (y,x,E) shape.

class hyperspy.io_plugins.bruker.Container
    Bases: object

class hyperspy.io_plugins.bruker.EDXSpectrum (spectrum)
    Bases: object
Wrap the objectified bruker EDS spectrum xml part to the python object, leaving all the xml and bruker clutter behind.

Arguments: spectrum – etree xml object, where spectrum.attrib[’Type’] should be ‘TRTSpectrum’

**energy_to_channel** *(energy, kV=True)*

convert energy to channel index, optional kwarg ‘kV’ (default: True) should be set to False if given energy units is in V.

```python
class hyperspy.io_plugins.bruker.HyperHeader(xml_str, indexes, instrument=None)
```

Wrap Bruker HyperMaping xml header into python object.

Arguments: xml_str – the uncompressed to be provided with extracted Header xml from bcf. indexes – list of indexes of available datasets

Methods: estimate_map_channels, estimate_map_depth

If Bcf is version 2, the bcf can contain stacks of hypermaps - thus header part can contain multiply sum eds spectras and it’s metadata per hypermap slice which can be selected using index. Bcf can record number of imagery from different imagining detectors (BSE, SEI, ARGUS, etc...): access to imagery is throught image index.

**calc_real_time** ()

calculate and return real time for whole hypermap in seconds

**estimate_map_channels** *(index=0)*

estimate minimal size of energy axis so any spectra from any pixel would not be truncated.

Arguments: index – index of the map if multiply hypermaps are present in the same bcf.

Returns: optimal channel number

**estimate_map_depth** *(index=0, downsample=1, for_numpy=False)*

estimate minimal dtype of array using cumulative spectra of the all pixels so that no data would be truncated.

Arguments: index – index of the hypermap if multiply hypermaps are present in the same bcf. (default 0)

downsample – downsample factor (should be integer; default 1) for_numpy – False produce unsigned, True signed (or unsigned) types:

if hypermap will be loaded using the pure python function where numpy’s inplace integer addition will be used – the dtype should be signed; if cython implementation will be used (default), then any returned dtypes can be safely unsigned. (default False)

Returns: numpy dtype large enough to use in final hypermap numpy array.

The method estimates the value from sum eds spectra, dividing the maximum energy pulse value from raster x and y and to be on the safe side multiplying by 2.

**gen_hspy_item_dict_basic** ()

**get_acq_instrument_dict** *(detector=False, **kwargs)*

return python dictionary with aquisition instrument mandatory data

**get_spectra_metadata** *(index=0)*

return objectified xml with spectra metadata Arguments: index – index of hypermap/spectra (default 0)
class hyperspy.io_plugins.bruker.SFSTreeItem(item_raw_string, parent)
    Bases: object

    Class to manage one internal sfs file.
    Reading, reading in chunks, reading and extracting, reading without extracting even if compression is present.
    Attributes: item_raw_string – the bytes from sfs file table describing the file parent – the item higher hierarchically in the sfs file tree
    Methods: read_piece, setup_compression_metadata, get_iter_and_properties, get_as_BytesIO_string

    get_as_BytesIO_string()
    Get the whole file as io.BytesIO object (in memory!).

    get_iter_and_properties()
    Generate and return the iterator of data chunks and properties of such chunks such as size and count.
    Method detects if data is compressed and uses iterator with decompression involved, else uses simple iterator of chunks.
    Returns (iterator, chunk_size, number_of_chunks)

    read_piece(offset, length)
    Read and returns raw byte string of the file without applying any decompression.
    Arguments: offset: seek value length: length of the data counting from the offset
    Returns: io.ByteIO object

    setup_compression_metadata()
    parse and setup the number of compression chunks
    and uncompressed chunk size as class attributes.
    Sets up attributes: self.uncompressed_blk_size, self.no_of_compr_blk

class hyperspy.io_plugins.bruker.SFS_reader(filename)
    Bases: object

    Class to read sfs file. SFS is AidAim software's(tm) single file system. The class provides basic reading capabilities of such container. It is capable to read compressed data in zlib, but SFS can contain other compression which is not implemented here. It is also not able to read encrypted sfs containers.
    This class can be used stand alone or inherited in construction of file readers using sfs technology.
    Attributes: filename
    Methods: get_file

    get_file(path)
    Return the SFSTreeItem (aka internal file) object from sfs container.
    Arguments: path – internal file path in sfs file tree. Path accepts only standard - forward slash for directories.
    Returns: object (SFSTreeItem), which can be read into byte stream, in chunks or whole using objects methods.
    Example: to get “file” object ‘kitten.png’ in folder ‘catz’ which resides in root directory of sfs, you would use:

    >>> instance_of_SFSReader.get_file('catz/kitten.png')

    See also: SFSTreeItem

3.1. hyperspy package
hyperspy.io_plugins.bruker.bcf_hyperspectra(obj_bcf, index=None, downsample=None, cutoff_at_kV=None, lazy=False)

Return hyperspy required list of dict with eds hyperspectra and metadata.

hyperspy.io_plugins.bruker.bcf_images(obj_bcf)

return hyperspy required list of dict with sem images and metadata.

hyperspy.io_plugins.bruker.bcf_reader(filename, select_type=None, index=None, downsample=1, cutoff_at_kV=None, instrument=None, lazy=False)

Reads a bruker bcf file and loads the data into the appropriate class, then wraps it into appropriate hyperspy required list of dictionaries used by hyperspy.api.load() method.

Keyword arguments:

select_type – One of: spectrum_image, image. If none specified, then function loads everything, else if specified, loads either just sem imagery, or just hyper spectral mapping data (default None).

index – index of dataset in bcf v2 can be None integer and ‘all’ (default None); None will select first available mapping if more than one. ‘all’ will return all maps if more than one present; integer will return only selected map.

downsample – the downsample ratio of hyperspectral array (downsampling height and width only), can be integer from 1 to inf, where ‘1’ means no downsampling will be applied (default 1).

cutoff_at_kV – if set (can be int of float >= 0) can be used either, to crop or enlarge energy range at max values. (default None)

instrument – str, either ‘TEM’ or ‘SEM’. Default is None.

hyperspy.io_plugins.bruker.dictionaryize(t)

hyperspy.io_plugins.bruker.file_reader(filename, *args, **kwds)

hyperspy.io_plugins.bruker.gen_detector_node(spectrum)

hyperspy.io_plugins.bruker.gen_elem_list(the_dict)

hyperspy.io_plugins.bruker.gen_iso_date_time(node)

hyperspy.io_plugins.bruker.get_mapping(mode)

hyperspy.io_plugins.bruker.guess_mode(hv)

there is no way to determine what kind of instrument was used from metadata: TEM or SEM. However simple guess can be made using the acceleration voltage, assuming that SEM is <= 30kV or TEM is >30kV

hyperspy.io_plugins.bruker.interpret(string)

interpret any string and return casted to appropriate dtype python object

hyperspy.io_plugins.bruker.parse_line(line_string)

standardize line description.

Bruker saves line description in all caps and omits the type if only one exists instead of using alfa

hyperspy.io_plugins.bruker.py_parse_hypermap(virtual_file, shape, dtype, downsample=1)

Unpack the Delphi/Bruker binary spectral map and return numpy array in memory efficient way using pure python implementation. (Slow!)

The function is long and complicated due to complexity of Delphi packed array. Whole parsing is placed in one function to reduce overhead of python function calls. For cleaner parsing logic, please, see fast cython implementation at hyperspy/io_plugins/unbcf_fast.pyx

The method is only meant to be used if for some reason c (generated with cython) version of the parser is not compiled.
virtual_file – virtual file handle returned by SFS_reader instance or by object inheriting it (e.g. BCF_reader instance)
shape – numpy shape
dtype – numpy dtype
downsamp;e – downsample factor

Note!: downsample, shape and dtype are interconnected and needs to be properly calculated otherwise wrong output or segfault is expected

NumPy array of bruker hypermap, with (y, x, E) shape.

```python
hyperspy.io_plugins.bruker.spx_reader(filename, lazy=False)
```

**hyperspy.io_plugins.dens module**

```python
hyperspy.io_plugins.dens.file_reader(filename, *args, **kwds)
```

**hyperspy.io_plugins.digital_micrograph module**

```python
class hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader(f)
Bases: object
Class to read Gatan Digital Micrograph (TM) files.
Currently it supports versions 3 and 4.
```

```python
dm_version, endian, tags_dict
parse_file, parse_header, get_image_dictionaries
check_data_tag_delimiter()
find_next_data_tag()
find_next_tag()
get_data_reader(enc_dtype)
get_image_dictionaries()
Returns the image dictionaries of all images in the file except the thumbnails.
```

Returns
```
Return type dict, None
```

```python
parse_array_definition()
Reads and returns the element type and length of the array.
The position in the file must be just after the array encoded dtype.
```

```python
parse_file()
```

```python
parse_header()
```

```python
parse_string_definition()
Reads and returns the length of the string.
The position in the file must be just after the string encoded dtype.
```

```python
parse_struct_definition()
Reads and returns the struct definition tuple.
The position in the file must be just after the struct encoded dtype.
```
parse_tag_group (size=False)
Parse the root TagGroup of the given DM3 file f. Returns the tuple (is_sorted, is_open, n_tags). endian can be either ‘big’ or ‘little’.

parse_tag_header ()

parse_tags (ntags, group_name='root', group_dict={})
Parse the DM file into a dictionary.

read_array (size, enc_eltype, extra=None, skip=False)
Read an array, defined by iarray, from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

property read_l_or_q
read_simple_data (etype)
Parse the data of the given DM3 file f with the given endianness (byte order). The infoArray iarray specifies how to read the data. Returns the tuple (file address, data). The tag data is stored in the platform’s byte order: ‘little’ endian for Intel, PC; ‘big’ endian for Mac, Motorola. If skip != 0 the data is actually skipped.

read_string (length, skip=False)
Read a string defined by the infoArray iarray from file f with a given endianness (byte order).
endian can be either ‘big’ or ‘little’.
If it’s a tag name, each char is 1-Byte; if it’s a tag data, each char is 2-Bytes Unicode,

read_struct (definition, skip=False)
Read a struct, defined by iarray, from file f with a given endianness (byte order). Returns a list of 2-tuples in the form (fieldAddress, fieldValue). endian can be either ‘big’ or ‘little’.

simple_type = (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
skipif4 (n=1)

class hyperspy.io_plugins.digital_micrograph.ImageObject (imdict, file, order='C', record_by=None)
Bases: object

property dtype
get_axes_dict ()
get_data ()
get_mapping ()
get_metadata (metadata={})
property intensity_calibration
property names
property offsets
property order
property record_by
property scales
property shape
property signal_type
property size
property title
property to_spectrum
property units
unpack_new_packed_complex(data)
unpack_packed_complex(tmpdata)

hyperspy.io_plugins.digital_micrograph.file_reader(filename, record_by=None, order=None, lazy=False, optimize=True)

Reads a DM3 file and loads the data into the appropriate class. data_id can be specified to load a given image within a DM3 file that contains more than one dataset.

Parameters

- record_by (str) – One of: SI, Signal2D
- order (str) – One of ‘C’ or ‘F’
- lazy (bool, default False) – Load the signal lazily.
- %s

hyperspy.io_plugins.edax module

hyperspy.io_plugins.edax.file_reader(filename, record_by='spectrum', endianess='<', **kwargs)

Parameters

- filename (str) – Name of file to read
- record_by (str) – EDAX EDS data is always recorded by ‘spectrum’, so this parameter is not used
- endianess (char) – Byte-order of data to read
- **kwargs – Additional keyword arguments supplied to the readers

hyperspy.io_plugins.edax.get_ipr_dtype_list(endianess='<', version=333)

Get the data type list for an IPR image description file. Further information about the file format is available here.

Table of header tags:

- version: 2 byte unsigned short; Current version number: 334
- imageType: 2 byte unsigned short; 0=empty; 1=electron; 2=xmap; 3=disk; 4=overlay
- label: 8 byte char array; Image label
- sMin: 2 byte unsigned short; Min collected signal
- sMax: 2 byte unsigned short; Max collected signal
- color: 2 byte unsigned short; color: 0=gray; 1=R; 2=G; 3=B; 4=Y; 5=M; 6=C; 8=overlay
- presetMode: 2 byte unsigned short; 0=clock; 1=live
- presetTime: 4 byte unsigned long; Dwell time for x-ray (millisecond)
- dataType: 2 byte unsigned short; 0=ROI; 1=Net intensity; 2=K ratio; 3=Wt%; 4=Mthin2
- timeConstantOld: 2 byte unsigned short; Amplifier pulse processing time (usec)

3.1. hyperspy package 345
• reserved1: 2 byte short; Not used
• roiStartChan: 2 byte unsigned short; ROI starting channel
• roiEndChan: 2 byte unsigned short; ROI ending channel
• userMin: 2 byte short; User Defined Min signal range
• userMax: 2 byte short; User Defined Max signal range
• iADC: 2 byte unsigned short; Electron detector number: 1; 2; 3; 4
• reserved2: 2 byte short; Not used
• iBits: 2 byte unsigned short; conversion type: 8; 12 (not used)
• nReads: 2 byte unsigned short; No. of reads per point
• nFrames: 2 byte unsigned short; No. of frames averaged (not used)
• fDwell: 4 byte float; Dwell time (not used)
• accV: 2 byte unsigned short; V_acc in units of 100V
• tilt: 2 byte short; Sample tilt [deg]
• takeoff: 2 byte short; Takeoff angle [deg]
• mag: 4 byte unsigned long; Magnification
• wd: 2 byte unsigned short; Working distance [mm]
• mppX: 4 byte float; Microns per pixel in X direction
• mppY: 4 byte float; Microns per pixel in Y direction
• nTextLines: 2 byte unsigned short; *No. of comment lines *
• charText: (4 x 32) byte character array; Comment text
• reserved3: 4 byte float; Not used
• nOverlayElements: 2 byte unsigned short; No. of overlay elements
• overlayColors: 16 array of 2 byte unsigned short; Overlay colors

# These two are specific to V334 of the file format, and are omitted # for compatibility with V333 of the
IPR format - timeConstantNew: 4 byte float; Amplifier time constant [usec] - reserved4: 2 array of 4 byte
float; Not used

Parameters

• endianess (char) – byte-order used to read the data
• version (float) – version of .ipr file to read (only 333 and 334 have been tested) Default
is 333 to be as backwards-compatible as possible, but the file version can be read from the
file anyway, so this parameter should always be set programmatically

Returns dtype_list – List of the data tags and data types that will be used by numpy to read an IPR
file.

Return type list

hyperspy.io_plugins.edax.get_spc_dtype_list (load_all=False, endianess=’<’, version=0.61)

Get the data type list for an SPC spectrum. Further information about the file format is available here.

Parameters
• **load_all**(bool) – Switch to control if all the data is loaded, or if just the important pieces of the signal will be read (speeds up loading time)

• **endianness**(char) – byte-order used to read the data

• **version**(float) – version of spc file to read (only 0.61 and 0.70 have been tested)
  Default is 0.61 to be as backwards-compatible as possible, but the file version can be read from the file anyway, so this parameter should always be set programmatically

• **of header tags**(Table) –
  
  – fVersion: 4 byte float; *File format Version*
  
  – aVersion: 4 byte float; *Application Version*
  
  – fileName: 8 array of 1 byte char; *File name w/o ‘.spc’ extension (OLD)*
  
  – collectDateYear: 2 byte short; *Year the spectrum was collected*
  
  – collectDateMon: 1 byte char; *Month the spectrum was collected*
  
  – collectTimeMin: 1 byte char; *Minute the spectrum was collected*
  
  – collectTimeHund: 1 byte char; *Hundredth second the spectrum was collected*
  
  – collectTimeSec: 1 byte char; *Second the spectrum was collected*
  
  – fileSize: 4 byte long; *Size of spectrum file in bytes*
  
  – dataStart: 4 byte long; *Start of spectrum data in bytes offset from 0 of file*
  
  – numPts: 2 byte short; *Number of spectrum pts*
  
  – intersectingDist: 2 byte short; *Intersecting distance * 100 (mm)*
  
  – workingDist: 2 byte short; *Working distance * 100*
  
  – scaleSetting: 2 byte short; *Scale setting distance * 100*
  
  – filler1: 24 byte;
  
  – spectrumLabel: 256 array of 1 byte char; *Type label for spectrum, 0-39=material type, 40-255=sample*
  
  – imageFilename: 8 array of 1 byte char; *Parent Image filename*
  
  – spotX: 2 byte short; *Spot X in parent image file*
  
  – spotY: 2 byte short; *Spot Y in parent image file*
  
  – imageADC: 2 byte short; *Image ADC value 0-4095*
  
  – discrValues: 5 array of 4 byte long; *Analyzer Discriminator Values*
  
  – discrEnabled: 5 array of 1 byte unsigned char; *Discriminator Flags (0=Disabled,1=Enabled)*
  
  – pileupProcessed: 1 byte char; *Pileup Processed Flag (0=No PU,1=Static PU, 2=Dynamic PU,...)*
  
  – fpgaVersion: 4 byte long; *Firmware Version.*
  
  – pileupProcVersion: 4 byte long; *Pileup Processing Software Version*
– NB5000CFG: 4 byte long; Defines Hitachi NB5000 Dual Stage Cfg 0=None, 10=Eucenic-
tric Crossx,11= Eucentric Surface 12= Side Entry - Side 13 = Side Entry - Top
– filler2: 12 byte;
– evPerChan: 4 byte long; EV/channel
– ADCTimeConstant: 2 byte short; ADC Time constant
– analysisType: 2 byte short; Preset mode 1=click, 2=count, 3=none, 4=live, 5=resume
– preset: 4 byte float; Analysis Time Preset value
– maxp: 4 byte long; Maximum counts of the spectrum
– maxPeakCh: 4 byte long; Max peak channel number
– xRayTubeZ: 2 byte short; XRF
– filterZ: 2 byte short; XRF
– current: 4 byte float; XRF
– sampleCond: 2 byte short; XRF Air= 0, Vacuum= 1, Helium= 2
– sampleType: 2 byte short; Bulk or thin
– xrayCollimator: 2 byte unsigned short; 0=None, 1=Installed
– xrayCapillaryType: 2 byte unsigned short; 0=Mono, 1=Poly
– xrayCapillarySize: 2 byte unsigned short; Range : 20 – 5000 Microns
– xrayFilterThickness: 2 byte unsigned short; Range : 0 – 10000 Microns
– spectrumSmoothed: 2 byte unsigned short; 1= Spectrum Smoothed, Else 0
– detector_Size_SiLi: 2 byte unsigned short; Eagle Detector 0=30mm, 1=80mm
– spectrumReCalib: 2 byte unsigned short; 1= Peaks Recalibrated, Else 0
– eagleSystem: 2 byte unsigned short; 0=None, 2=Eagle2, 3=Eagle3, 4-Xscope
– sumPeakRemoved: 2 byte unsigned short; 1= Sum Peaks Removed, Else 0
– edaxSoftwareType: 2 byte unsigned short; 1= Team Spectrum, Else 0
– filler3: 6 byte;
– escapePeakRemoved: 2 byte unsigned short; 1=Escape Peak Was Removed, Else 0
– analyzerType: 4 byte unsigned long; Hardware type 1=EDI1, 2=EDI2, 3=DPP2,
  31=DPP-FR, 32=DPP-FR2, 4=DPP3, 5= APOLLO XLT/XLS/DPP-4 (EDPP)
– startEnergy: 4 byte float; Starting energy of spectrum
– endEnergy: 4 byte float; Ending energy of spectrum
– liveTime: 4 byte float; LiveTime
– tilt: 4 byte float; Tilt angle
– takeoff: 4 byte float; Take off angle
– beamCurFact: 4 byte float; Beam current factor
– detReso: 4 byte float; Detector resolution
- `detectType`: 4 byte unsigned long; *Detector Type*: 1=Std-BE, 2=UTW, 3=Super UTW, 4=ECON 3/4 Open, 5=ECON 3/4 Closed, 6=ECON 5/6 Open, 7=ECON 5/6 Closed, 8=TEMECON; Add + 10 For Sapphire SiLi Detectors, (11-18), which started shipping in 1996. 30 = APOLLO 10 SDD, 31=APOLLO XV, 32 = APOLLO 10+, 40 = APOLLO 40 SDD, 50 = APOLLO-X, 51=APOLLO-XP, 52 = APOLLO-XL, 53 = APOLLO XL-XRF, 60 =APOLLO-XLT-LS, 61 =APOLLO-XLT-NW, 62 =APOLLO-XLT-SUTW

- `parThick`: 4 byte float; *Parlodion light shield thickness*

- `alThick`: 4 byte float; *Aluminum light shield thickness*

- `beWinThick`: 4 byte float; *Be window thickness*

- `auThick`: 4 byte float; *Gold light shield thickness*

- `siDead`: 4 byte float; *Si dead layer thickness*

- `siLive`: 4 byte float; *Si live layer thickness*

- `xrayInc`: 4 byte float; *X-ray incidence angle*

- `azimuth`: 4 byte float; *Azimuth angle of detector*

- `elevation`: 4 byte float; *Elevation angle of detector*

- `bCoeff`: 4 byte float; *K-line B coefficient*

- `cCoeff`: 4 byte float; *K-line C coefficient*

- `tailMax`: 4 byte float; *Tail function maximum channel*

- `tailHeight`: 4 byte float; *Tail height adjustment percentage*

- `kV`: 4 byte float; *Acc voltage*

- `apThick`: 4 byte float; *Ap window thickness*

- `xTilt`: 4 byte float; *x tilt angle for mDX*

- `yTilt`: 4 byte float; *y tilt angle for mDX*

- `yagStatus`: 4 byte unsigned long; 0 = N/A, 1 = YAG OUT, 2 = YAG IN

- `filler4`: 24 byte;

- `rawDataType`: 2 byte unsigned short; *TEM or SEM data*

- `totalBkgdCount`: 4 byte float; *Accumulated background counts*

- `totalSpectralCount`: 4 byte unsigned long; *Accumulated spectrum counts*

- `avgInputCount`: 4 byte float; *Average spectral counts*

- `stdDevInputCount`: 4 byte float; *Standard deviation of spectral counts*

- `peakToBack`: 2 byte unsigned short; *Peak to background setting*. 0 = off, 1 = low, 2 = medium, 3 = high, 4 = user selected

- `peakToBackValue`: 4 byte float; *Peak to back value*

- `filler5`: 38 byte;

- `numElem`: 2 byte short; *Number of peak id elements 0-48*

- `at`: 48 array of 2 byte unsigned short; *atomic numbers for peak id elems*

- `line`: 48 array of 2 byte unsigned short; *line numbers for peak id elems*

- `energy`: 48 array of 4 byte float; *float energy of identified peaks*
- height: 48 array of 4 byte unsigned long; *height in counts of id'ed peaks*
- spkht: 48 array of 2 byte short; *sorted peak height of id'ed peaks*
- filler5_1: 30 byte;
- numRois: 2 byte short; *Number of ROI's defined 0-48*
- st: 48 array of 2 byte short; *Start channel # for each ROI*
- end: 48 array of 2 byte short; *End channel # for each ROI*
- roiEnable: 48 array of 2 byte short; *ROI enable/disable flags*
- roiNames: (24 x 8) array of 1 byte char; *8 char name for eah ROI*
- filler5_2: 1 byte;
- userID: 80 array of 1 byte char; *User ID (Vision S/W) - Overlapping*
- filler5_3: 111 byte;
- sRoi: 48 array of 2 byte short; *sorted ROI heights*
- scaNum: 48 array of 2 byte short; *SCA number assigned for each ROI*
- filler6: 12 byte;
- backgrdWidth: 2 byte short; *Background width*
- manBkgrdPerc: 4 byte float; *Percentage to move manual background down*
- numBkgrdPts: 2 byte short; *Number of background points (2-64)*
- backMethod: 4 byte unsigned long; *Background method 1=auto, 2=manual*
- backStEng: 4 byte float; *Starting energy of background*
- backEndEng: 4 byte float; *Ending energy of background*
- bg: 64 array of 2 byte short; *Channel # of background point*
- bgType: 4 byte unsigned long; *Background type. 1 = curve, 2 = linear.*
- concenKev1: 4 byte float; *First concentration background point*
- concenKev2: 4 byte float; *Second concentration background point*
- concenMethod: 2 byte short; 0 = Off, 1 = On
- jobFilename: 32 array of 1 byte char; *Vision Job Filename*
- filler7: 16 byte;
- numLabels: 2 byte short; *Number of displayed labels*
- label: (10 x 32) array 1 byte char; *32 character labels on the spectrum*
- labelx: 10 array of 2 byte short; *x position of label in terms of channel #*
- labely: 10 array of 4 byte long; *y position of label in terms of counts*
- zListFlag: 4 byte long; *Flag to indicate if Z List was written*
- bgPercentes: 64 array of 4 byte float; *Percentage to move background point up and down.*
- IswGBg: 2 byte short; = 1 if new backgrd pts exist
- BgPoints: 5 array of 4 byte float; *Background points*
- IswGConc: 2 byte short; = 1 if given concentrations exist
– numConcen: 2 byte short; Number of elements (up to 24)
– ZList: 24 array of 2 byte short; Element list for which given concentrations exist
– GivenConc: 24 array of 4 byte float; Given concentrations for each element in Zlist
– filler8: 598 byte;
– s: 4096 array of 4 byte long; counts for each channel
– longFileName: 256 array of 1 byte char; Long filename for 32 bit version
– longImageFileName: 256 array of 1 byte char; Associated long image file name
– ADCTimeConstantNew: 4 byte float; Time constant: 2.5...100 OR 1.6...102.4 us
# the following datatypes are only included for version 0.70:
– filler9: 60 byte;
– numZElements: 2 byte short; number of Z List elements for quant
– zAtoms: 48 array of 2 byte short; Z List Atomic numbers
– zShells: 48 array of 2 byte short; Z List Shell numbers

Returns dtype_list – List of the data tags and data types that will be used by numpy to read an SPC file header.

Return type list

hyperspy.io_plugins.edax.get_spd_dtype_list(endianess='<')
Get the data type list for an SPD map. Further information about the file format is available here.

Table of header tags:

- tag: 16 byte char array; File ID tag (“MAPSPECTRA_DATA”)
- version: 4 byte long; File version
- nSpectra: 4 byte long; Number of spectra in file
- nPoints: 4 byte long; Number of map pixels in X direction
- nLines: 4 byte long; Number of map pixels in Y direction
- nChannels: 4 byte long; Number of channels per spectrum
- countBytes: 4 byte long; Number of count bytes per channel
- dataOffset: 4 byte long; File offset in bytes for data start
- nFrames: 4 byte long; Number of frames in live spectrum mapping
- fName: 120 byte char array; File name of electron image acquired during mapping

Parameters endianess (byte-order used to read the data) –

Returns dtype_list – List of the data tags and data types that will be used by numpy to read an SPC file header.

Return type list

hyperspy.io_plugins.edax.spc_reader(filename, endianess='<', load_all_spç=False, **kwargs)
Read data from an SPC spectrum specified by filename.

Parameters
**filename** *(str)* – Name of SPC file to read

**endianess** *(char)* – Byte-order of data to read

**load_all_spc** *(bool)* – Switch to control if all of the .spc header is read, or just the important parts for import into HyperSpy

**kwargs** – Remaining arguments are passed to the Numpy memmap function

Returns list with dictionary of signal information to be passed back to hyperspy.io.load_with_reader

Return type list

---

**filename** *(str)* – Name of SPD file to read

**endianess** *(char)* – Byte-order of data to read

**spc_fname** *(None or str)* – Name of file from which to read the spectral calibration. If data was exported fully from EDAX TEAM software, an .spc file with the same name as the .spd should be present. If None, the default filename will be searched for. Otherwise, the name of the .spc file to use for calibration can be explicitly given as a string.

**ipr_fname** *(None or str)* – Name of file from which to read the spatial calibration. If data was exported fully from EDAX TEAM software, an .ipr file with the same name as the .spd (plus a “_Img” suffix) should be present. If None, the default filename will be searched for. Otherwise, the name of the .ipr file to use for spatial calibration can be explicitly given as a string.

**load_all_spc** *(bool)* – Switch to control if all of the .spc header is read, or just the important parts for import into HyperSpy

**kwargs** – Remaining arguments are passed to the Numpy memmap function

Returns list with dictionary of signal information to be passed back to hyperspy.io.load_with_reader

Return type list

---

**signals** *(dictionary)* – Dictionary which contains all datasets as Signal instances.

**user** *(None)*

**microscope** *(None)*

**sample** *(None)*

**comments** *(None)*

**log_info()**

Class for storing electron microscopy datasets.

The EMD class can hold an arbitrary amount of datasets in the signals dictionary. These are saved as HyperSpy Signal instances. Global metadata are saved in four dictionaries (user, microscope, sample, comments). To print relevant information about the EMD instance use the log_info() function. EMD instances can be loaded from and saved to emd-files, an hdf5 standard developed at Lawrence Berkeley National Lab (https://emd.datasets.com/).

**signals**

Dictionary which contains all datasets as Signal instances.

**Type** dictionary
user
Dictionary which contains user related metadata.
Type dictionary

microscope
Dictionary which contains microscope related metadata.
Type dictionary

sample
Dictionary which contains sample related metadata.
Type dictionary

comments
Dictionary which contains additional commentary metadata.
Type dictionary

add_signal(signal, name=None, metadata=None)
Add a HyperSpy signal to the EMD instance and make sure all metadata is present.

Parameters

• signal (Signal) – HyperSpy signal which should be added to the EMD instance.
• name (string, optional) – Name of the (used as a key for the signals dictionary).
  If not specified, signal.metadata.General.title will be used. If this is an empty string, both
  name and signal title are set to ‘dataset’ per default. If specified, name overwrites the
  signal title.
• metadata (dictionary) – Dictionary which holds signal specific metadata which
  will be added to the signal.

Returns

Return type None

Notes
This is the preferred way to add signals to the EMD instance. Directly adding to the signals dictionary
is possible but does not make sure all metadata are correct. This method is also called in the standard
constructor on all entries in the signals dictionary!

classmethod load_from_emd(filename, lazy=False, dataset_name=None)
Construct EMD object from an emd-file.

Parameters

• filename (string) – The name of the emd-file from which to load the signals. Standard
  format is ‘*.emd’.
• False (bool, optional) – If False (default) loads data to memory. If True, enables
  loading only if requested.
• dataset_name (string or iterable, optional) – Only add dataset with
  specific name. Note, this has to be the full group path in the file. For example ‘/ex-
  perimental/science_data’. If the dataset is not found, an IOError with the possible datasets
  will be raised. Several names can be specified in the form of a list.

Returns emd – A EMD object containing the loaded signals.

Return type EMD
log_info()

(all relevant information about the EMD instance.

save_to_emd(filename='datacollection.emd')

Save EMD data in a file with emd(hdf5)-format.

Parameters

filename (string, optional) -- The name of the emd-file in which to store the signals. The default is `datacollection.emd`.

Returns

Return type None

class hyperspy.io_plugins.emd.FeiEMDReader(filename, select_type=None, first_frame=0, last_frame=None, sum_frames=True, sum_EDS_detectors=True, rebin_energy=1, SI_dtype=None, load_SI_image_stack=False, lazy=False)

Bases: object

Class for reading FEI electron microscopy datasets.

The FeiEMDReader reads EMD files saved by the FEI Velox software package.

dictionaries

List of dictionaries which are passed to the file_reader.

Type list

im_type

String specifying whether the data is an image, spectrum or spectrum image.

Type string

class hyperspy.io_plugins.emd.FeiSpectrumStream(stream_group, reader)

Bases: object

Read spectrum image stored in FEI’s stream format

Once initialized, the instance of this class supports numpy style indexing and slicing of the data stored in the stream format.

get_pixelsize_offset_unit()

property shape

stream_to_array(stream_data, spectrum_image=None)

Convert stream to array.

Parameters

• stream_data (array) --

• spectrum_image (array or None) -- If array, the data from the stream are added to the array. Otherwise it creates a new array and returns it.

stream_to_sparse_array(stream_data)

Convert stream in sparse array

Parameters stream_data (array) --

hyperspy.io_plugins.emd.calculate_chunks(shape, dtype, chunk_size_mb=100)

Calculate chunks to get target chunk size.

The chunks are optimized for C-order reading speed.

Parameters
• **shape** *(tuple of ints)* – The shape of the array
• **dtype** *(string or numpy dtype)* – The dtype of the array
• **chunk_size_mb** *(int)* – The maximum size of the resulting chunks in MB. The default is 100MB as recommended by the dask documentation.

```python
hyperspy.io_plugins.emd.fei_check(filename)
```

Function to check if the EMD file is an FEI file.

**Parameters**

- **filename** *(string)* – The name of the emd-file from which to load the signals. Standard format is ‘*.emd’.

**Returns**

- **Return type** Boolean

```python
hyperspy.io_plugins.emd.file_reader(filename, log_info=False, lazy=False, **kwds)
```

```python
hyperspy.io_plugins.emd.file_writer(filename, signal, signal_metadata=None, user=None, microscope=None, sample=None, comments=None, **kwds)
```

### hyperspy.io_plugins.fei module

```python
hyperspy.io_plugins.fei.convert_xml_to_dict(xml_object)
```

```python
hyperspy.io_plugins.fei.dimension_array_dtype(n, DescriptionLength, UnitsLength)
```

```python
hyperspy.io_plugins.fei.emi_reader(filename, dump_xml=False, **kwds)
```

```python
hyperspy.io_plugins.fei.emixml2dtb(et, dictree)
```

```python
hyperspy.io_plugins.fei.file_reader(filename, *args, **kwds)
```

```python
hyperspy.io_plugins.fei.get_axes_from_position(header, data)
```

```python
hyperspy.io_plugins.fei.get_calibration_from_position(position)
```

This function assumes rastering on a regular grid for the full size of each dimension before rastering over another one. For example: a11, a12, a13, a21, a22, a23 for a 2x3 grid.

**Parameters**

- **position** *(numpy array.)* – Position coordinates of the axis. Normally as in PositionX/Y of the ser file.

**Returns**

- **axis_attr** dictionary with *size*, *scale*, *offset* keys.

```python
hyperspy.io_plugins.fei.get_data_dtype_list(file, offset, record_by)
```

```python
hyperspy.io_plugins.fei.get_data_tag_dtype_list(data_type_id)
```

```python
hyperspy.io_plugins.fei.get_header_dtype_list(file)
```

```python
hyperspy.io_plugins.fei.get_lengths(file)
```

```python
hyperspy.io_plugins.fei.get_xml_info_from_emi(emi_file)
```

```python
hyperspy.io_plugins.fei.guess_record_by(record_by_id)
```

```python
hyperspy.io_plugins.fei.load_only_data(filename, array_shape, record_by, num_axes, data=None, header=None, only_valid_data=False)
```

```python
hyperspy.io_plugins.fei.load_ser_file(filename)
```

3.1. hyperspy package
hyperspy.io_plugins.fei.log_struct_array_values(struct_array)
hyperspy.io_plugins.fei.parse_ExperimentalDescription(et, dictree)
hyperspy.io_plugins.fei.parse_TrueImageHeaderInfo(et, dictree)
hyperspy.io_plugins.fei.readLELong(file)
    Read 4 bytes as little endian integer in file
hyperspy.io_plugins.fei.readLELongLong(file)
    Read 8 bytes as little endian integer in file
hyperspy.io_plugins.fei.readLEShort(file)
    Read 2 bytes as little endian integer in file
hyperspy.io_plugins.fei.ser_reader(filename, objects=None, lazy=False, only_valid_data=False)
    Reads the information from the file and returns it in the HyperSpy required format.

hyperspy.io_plugins.hspy module

hyperspy.io_plugins.hspy.dict2hdfgroup(dictionary, group, **kwds)
hyperspy.io_plugins.hspy.file_reader(filename, backing_store=False, lazy=False, **kwds)
hyperspy.io_plugins.hspy.file_writer(filename, signal, *args, **kwds)
hyperspy.io_plugins.hspy.get_hspy_format_version(f)
hyperspy.io_plugins.hspy.get_signal_chunks(shape, dtype, signal_axes=None)
    Function that calculates chunks for the signal, preferably at least one chunk per signal space.
    Parameters
    • shape (tuple) – the shape of the dataset to be stored / chunked
    • dtype((dtype, string)) – the numpy dtype of the data
    • signal_axes((None, iterable of ints)) – the axes defining “signal space” of
      the dataset. If None, the default h5py chunking is performed.

hyperspy.io_plugins.hspy.hdfgroup2dict(group, dictionary=None, lazy=False)
hyperspy.io_plugins.hspy.hdfgroup2signaldict(group, lazy=False)
hyperspy.io_plugins.hspy.overwrite_dataset(group, data, key, signal_axes=None, chunks=None, **kwds)
hyperspy.io_plugins.hspy.write_signal(signal, group, **kwds)

hyperspy.io_plugins.image module

hyperspy.io_plugins.image.file_reader(filename, **kwds)
    Read data from any format supported by PIL.
    Parameters filename(str) –

hyperspy.io_plugins.image.file_writer(filename, signal, file_format='png', **kwds)
    Writes data to any format supported by PIL
    Parameters
    • filename(str) –
• **signal** *(a Signal instance)*

• **file_format** *(str)* – The fileformat defined by its extension that is any one supported by PIL.

**hyperspy.io_plugins.mrc module**

hyperspy.io_plugins.mrc.file_reader(filename, endianess='<', **kwds)

hyperspy.io_plugins.mrc.get_data_type(index, endianess='<')

hyperspy.io_plugins.mrc.get_fei_dtype_list(endianess='<')

hyperspy.io_plugins.mrc.get_std_dtype_list(endianess='<')

**hyperspy.io_plugins.mrcz module**

**hyperspy.io_plugins.msa module**

hyperspy.io_plugins.msa.file_reader(filename, encoding='latin-1', **kwds)

hyperspy.io_plugins.msa.file_writer(filename, signal, format=None, separator=' ', encoding='latin-1')

hyperspy.io_plugins.msa.parse_msa_string(string, filename=None)

Parse an EMSA/MSA file content.

Parameters

• **string** *(string or file object)* – It must comply with the EMSA/MSA standard.

• **filename** *(string or None)* – The filename.

• **Returns** –

• **file_data_list** *(list)* – The list contains a dictionary that contains the parsed information. It can be used to create a :class:`BaseSignal` using :func:`hyperspy.io.dict2signal`.

**hyperspy.io_plugins.netcdf module**

hyperspy.io_plugins.netcdf.file_reader(filename, *args, **kwds)

hyperspy.io_plugins.netcdf.nc_hyperspy_reader_0dot1(ncfile, filename, *args, **kwds)

**hyperspy.io_plugins.protochips module**

class hyperspy.io_plugins.protochips.ProtochipsCSV(filename)

    Bases: object

    get_dictionary(quantity)

hyperspy.io_plugins.protochips.file_reader(filename, *args, **kwds)
hyperspy.io_plugins.ripple module

**hyperspy.io_plugins.ripple.correct_INCA_format** *(fp)*

**hyperspy.io_plugins.ripple.file_reader** *(filename, rpl_info=None, encoding='latin-1', mmap_mode='c', *args, **kwds)*

Parses a Lispix (http://www.nist.gov/lispix/) ripple (.rpl) file and reads the data from the corresponding raw (.raw) file; or, read a raw file if the dictionary rpl_info is provided.

This format is often uses in EDS/EDX experiments.

Images and spectral images or data cubes that are written in the (Lispix) raw file format are just a continuous string of numbers.

Data cubes can be stored image by image, or spectrum by spectrum. Single images are stored row by row, vector cubes are stored row by row (each row spectrum by spectrum), image cubes are stored image by image.

All of the numbers are in the same format, such as 16 bit signed integer, IEEE 8-byte real, 8-bit unsigned byte, etc.

The “raw” file should be accompanied by text file with the same name and “.rpl” extension. This file lists the characteristics of the raw file so that it can be loaded without human intervention.

Alternatively, dictionary ‘rpl_info’ containing the information can be given.

Some keys are specific to HyperSpy and will be ignored by other software.

RPL stands for “Raw Parameter List”, an ASCII text, tab delimited file in which HyperSpy reads the image parameters for a raw file.

**TABLE OF RPL PARAMETERS**

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>int</td>
<td># pixels per row height</td>
</tr>
<tr>
<td>height</td>
<td>int</td>
<td># number of rows depth</td>
</tr>
</tbody>
</table>
| depth          | int      | # number of images or spectral pts offset int # bytes to skip data-type str # ‘signed’, ‘unsigned’, or ‘float’ data-length str # bytes per pixel ‘1’, ‘2’, ‘4’, or ‘8’ byte-order str # ‘big-endian’, ‘little-endian’, or ‘dont-care’ record-by str # ‘image’, ‘vector’, or ‘dont-care’ X-ray keys: ev-per-chan int # optional, eV per channel detector-peak-width-ev int # optional, FWHM for the Mn K-alpha line HyperSpy-specific keys depth-origin int # energy offset in pixels depth-scale float # energy scaling (units per pixel) depth-units str # energy units, usually eV depth-name str # Name of the magnitude stored as depth width-origin int # column offset in pixels width-scale float # column scaling (units per pixel) width-units str # column units, usually nm width-name str # Name of the magnitude stored as width height-origin int # row offset in pixels height-scale float # row scaling (units per pixel) height-units str # row units, usually nm height-name str # Name of the magnitude stored as height signal str # Type of the signal stored, e.g. EDS_SEM convergence-angle float # TEM convergence angle in mrad collection-angle float # EELS spectrometer collection semi-angle in mrad beam-energy float # TEM beam energy in keV elevation-angle float # Elevation angle of the EDS detector azimuth-angle float # Elevation angle of the EDS detector live-time float # Live time per spectrum energy-resolution float # Resolution of the EDS (FWHM of MnKa) tilt-stage float # The tilt of the stage date str # date in ISO 8601 time str # time in ISO 8601 title str # title of the signal to be stored

**NOTES**

When ‘data-length’ is 1, the ‘byte order’ is not relevant as there is only one byte per datum, and ‘byte-order’ should be ‘dont-care’.

When ‘depth’ is 1, the file has one image, ‘record-by’ is not relevant and should be ‘dont-care’. For spectral images, ‘record-by’ is ‘vector’. For stacks of images, ‘record-by’ is ‘image’.
Floating point numbers can be IEEE 4-byte, or IEEE 8-byte. Therefore if data-type is float, data-length MUST be 4 or 8.

The rpl file is read in a case-insensitive manner. However, when providing a dictionary as input, the keys MUST be lowercase.

Comment lines, beginning with a semi-colon ‘;’ are allowed anywhere.

The first non-comment in the rpl file line MUST have two column names: ‘name_1’<TAB>’name_2’; any name would do e.g. ‘key’<TAB>’value’.

Parameters can be in ANY order.

In the rpl file, the parameter name is followed by ONE tab (spaces are ignored) e.g.: ‘data-length’<TAB>’2’

In the rpl file, other data and more tabs can follow the two items on each row, and are ignored.

Other keys and values can be included and are ignored.

Any number of spaces can go along with each tab.

hyperspy.io_plugins.ripple.file_writer(filename, signal, encoding='latin-1', *args, **kwds)

hyperspy.io_plugins.ripple.parse_ripple(fp)

hyperspy.io_plugins.ripple.read_raw(rpl_info, fp, mmap_mode='c')
Read the raw file object ‘fp’ based on the information given in the 'rpl_info' dictionary.

Parameters

- rpl_info(dict) – A dictionary containing the keywords as parsed by read_rpl
- fp –
- mmap_mode((None, 'r+', 'r', 'w+', 'c'), optional) –
- not None, then memory-map the file, using the given mode(If)–
- numpy.memmap) The mode has no effect for pickled or((see)–
- files.(zipped)–
- memory-mapped array is stored on disk, and not directly loaded(A)–
- memory. However, it can be accessed and sliced like any (into)
- Memory mapping is especially useful for accessing(ndarray.)–
- fragments of large files without reading the entire file (small)–
- memory.(into)–

hyperspy.io_plugins.ripple.write_raw(filename, signal, record_by)
Writes the raw file object

filename [string] the filename, either with the extension or without it
record_by [string] ‘vector’ or ‘image’

hyperspy.io_plugins.ripple.write_rpl(filename, keys_dictionary, encoding='ascii')
class hyperspy.io_plugins.semper_unf.SemperFormat(data, title=", offsets=(0.0, 0.0, 0.0), scales=(1.0, 1.0, 1.0), units=(<undefined>, <undefined>, <undefined>), metadata=None)

Bases: object

Class for importing and exporting SEMPER .unf-files.

The **SemperFormat** class represents a SEMPER binary file format with a header, which holds additional information. .unf-files can be saved and read from files.

**data**

The phase map or magnetization information in a 3D array (with one slice).

  * Type ndarray (N=3)

**title**

Title of the file (not to be confused with the filename).

  * Type string

**offsets**

Offset shifts (in nm) of the grid origin (does not have to start at 0) in x, y, z.

  * Type tuple (N=3) of floats

**scales**

Grid spacing (nm per pixel) in x, y, z.

  * Type tuple (N=3) of floats

**units**

Units of the grid in x, y, z.

  * Type tuple (N=3) of strings

**metadata**

A dictionary of all flags and metadata present in the .unf-file.

  * Type dictionary

**HEADER_DTYPES** = [('NCOL', '<i2'), ('NROW', '<i2'), ('NLAY', '<i2'), ('ICLASS', '<i2'), ('IFORM', '<i2'), ('IFLAG', '<i2'), ('IFORM', '<i2')


**ICLASS_DICT_INV** = {'image': 1, 'macro': 2, 'fourier': 3, 'spectrum': 4, 'correlation': 5, <undefined>: 6, 'walsh': 7, 'position list': 8, 'histogram': 9, 'display look-up table': 10}

**IFORM_DICT** = {0: <class 'numpy.int8'>, 1: <class 'numpy.int16'>, 2: <class 'numpy.float32'>, 3: <class 'numpy.complex64'>, 4: <class 'numpy.int32'>}

**IFORM_DICT_INV** = {<class 'numpy.int8'>: 0, <class 'numpy.int16'>: 1, <class 'numpy.float32'>: 2, <class 'numpy.complex64'>: 3, <class 'numpy.int32'>: 4}

**LABEL_DTYPES** = [('SEMPER', ('<i2', 6)), ('NCOL', ('<i2', 2)), ('NROW', ('<i2', 2)), ('SEMPER', ('<i2', 6))]

**classmethod from_signal**(signal)

Import a **SemperFormat** object from a **Signal** object.

  * Parameters signal (Signal) – The signal which should be imported.

  * Returns 

  * Return type None

**classmethod load_from_unf**(filename, lazy=False)

Load a .unf-file into a **SemperFormat** object.
Parameters `filename (string)` – The name of the unf-file from which to load the data. Standard format is '*.unf'.

Returns `semper` – SEMPER file format object containing the loaded information.

Return type `SemperFormat` (N=1)

`log_info()`

log important flag information of the `SemperFormat` object.

Parameters None –

Returns

Return type None

`save_to_unf(filename='semper.unf', skip_header=False)`

Save a `SemperFormat` to a file.

Parameters

• `filename (string, optional) –` The name of the unf-file to which the data should be written.

• `skip_header (boolean, optional) –` Determines if the header, title and label should be skipped (useful for some other programs). Default is False.

Returns

Return type None

`to_signal(lazy=False)`

Export a `SemperFormat` object to a `Signal` object.

Parameters None –

Returns `signal` – The exported signal.

Return type `Signal`

`hyperspy.io_plugins.semper_unf.file_reader(filename, **kwds)`

`hyperspy.io_plugins.semper_unf.file_writer(filename, signal, **kwds)`

`hyperspy.io_plugins.semper_unf.pack_to_intbytes(fmt, value)`

Pack a value into a byte list using format `fmt` and represent it as int (range 0-255).

`hyperspy.io_plugins.semper_unf.unpack_from_intbytes(fmt, byte_list)`

Read in a list of bytes (as int with range 0-255) and unpack them with format `fmt`.

`hyperspy.io_plugins.tiff.file_reader(filename, record_by='image', force_read_resolution=False, **kwds)`

Read data from tif files using Christoph Gohlke’s tifffile library. The units and the scale of images saved with ImageJ or Digital Micrograph is read. There is limited support for reading the scale of files created with Zeiss and FEI SEMs.

Parameters

• `filename (str)` –

• `record_by ("image")` – Has no effect because this format only supports recording by image.
• `force_read_resolution` (*bool*) – Default: False. Force reading the x_resolution, y_resolution and the resolution_unit of the tiff tags. See [http://www.awaresystems.be/imaging/tiff/tifftags/resolutionunit.html](http://www.awaresystems.be/imaging/tiff/tifftags/resolutionunit.html)

• optional(**kwds,*) –

`hyperspy.io_plugins.tiff.file_writer` *(filename, signal, export_scale=True, extratags=[], **kwds)*

Writes data to tif using Christoph Gohlke’s tifffile library

**Parameters**

• `filename` (*str*) –

• `signal` (*a BaseSignal instance*) –

• `export_scale` (*bool*) – default: True Export the scale and the units (compatible with DM and ImageJ) to appropriate tags.

`hyperspy.io_plugins.tiff.get_metadata_mapping` *(tiff_page, op)*

`hyperspy.io_plugins.tiff.get_tvips_mapping` *(mapped_magnification)*

---

**hyperspy.io_plugins.unbcf_fast module**

**Module contents**

**hyperspy.learn package**

**Submodules**

**hyperspy.learn.mlpca module**

`hyperspy.learn.mlpca.mlpca` *(X, varX, p, convlim=1e-10, maxiter=50000, fast=False)*

This function performs MLPCA with missing data.

**Parameters**

• `X` (*numpy array*) – is the mxn matrix of observations.

• `stdX` (*numpy array*) – is the mxn matrix of standard deviations associated with X (zeros for missing measurements).

• `p` (*int*) – The model dimensionality.

**Returns**

• `U, S, V` (*numpy array*) – are the pseudo-svd parameters.

• `Sobj` (*numpy array*) – is the value of the objective function.

• `ErrFlag` (*{0, 1}* ) – indicates exit conditions: 0 = nkmal termination 1 = max iterations exceeded.

**hyperspy.learn.mva module**

class `hyperspy.learn.mva.LearningResults`

    Bases: `object`

    `bss_algorithm` = None
```python
bss_factors = None
bss_loadings = None
centre = None
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) – If the decomposition results are lazy, also compute the results. Default is False.

decomposition_algorithm = None
explained_variance = None
explained_variance_ratio = None
factors = None
load(filename)
    Load the results of a previous decomposition and demixing analysis from a file.

Parameters filename (string) –

loadings = None
mean = None
navigation_mask = None
number_significant_components = None
original_shape = None
output_dimension = None
poissonian_noise_normalized = None
save (filename, overwrite=None)
    Save the result of the decomposition and demixing analysis

Parameters

• filename (string) –
• overwrite ((True, False, None)) – If True (False) overwrite (don’t overwrite) the file if it exists. If None (default), ask what to do if file exists.

signal_mask = None
summary()
    Prints a summary of the decomposition and demixing parameters to the stdout

unfolded = None
unmixing_matrix = None

class hyperspy.learn.mva.MVA
    Bases: object

    Multivariate analysis capabilities for the Signal1D class.
```
**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', compute=False, **kwargs)

Blind source separation (BSS) on the result on the decomposition.

Available algorithms: FastICA, JADE, CuBICA, and TDSEP

**Parameters**

- **number_of_components** (int) – number of principal components to pass to the BSS algorithm
- **algorithm** ([FastICA, JADE, CuBICA, TDSEP]) – BSS algorithms available.
- **diff_order** (int) – 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 or list of ints or strings) – If None, when diff_order is greater than 1 and signal_dimension (navigation_dimension) when on_loadings is False (True) is greater than 1, the differences are calculated across all signal (navigation) axes. Otherwise the axes can be specified in a list.
- **factors** (Signal or numpy array.) – 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** (boolen numpy array) – choose the components to use by the boolean list. It permits to choose non contiguous components.
- **mask** (bool numpy array or Signal instance.) – 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) – If True, perform the BSS on the loadings of a previous decomposition. If False, performs it on the factors.
- **reverse_component_criterion** (str {'factors', 'loadings'}) – Use either the factor or the loading to determine if the component needs to be reversed.
- **compute** (bool) – If the decomposition results are lazy, compute the BSS components so that they are not lazy. Default is False.
- ****kwargs** (extra key word arguments) – Any keyword arguments are passed to the BSS algorithm.
- **documentation is here, with more arguments that can be passed as **kwargs**(FastICA)=**

**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, polyfit=None, reproject=None, return_info=False, **kwargs)

Decomposition with a choice of algorithms

The results are stored in self.learning_results

**Parameters**

- **normalize_poissonian_noise** (bool) – If True, scale the SI to normalize Poissonian noise
• **algorithm** (`'svd' | 'fast_svd' | 'mlpca' | 'fast_mlpca' | 'nmf' | 'sparse_pca' | 'mini_batch_sparse_pca' | 'RPCA_GoDec' | 'ORPCA')
• **output_dimension** (`None or int`) - number of components to keep/calculate
• **centre** (`None | 'variables' | 'trials'`) - If None no centring is applied. If 'variable' the centring will be performed in the variable axis. If 'trials', the centring will be performed in the 'trials' axis. It only has effect when using the svd or fast_svd algorithms
• **auto_transpose** (`bool`) - If True, automatically transposes the data to boost performance. Only has effect when using the svd or fast_svd algorithms.
• **navigation_mask** (`boolean numpy array`) - The navigation locations marked as True are not used in the decomposition.
• **signal_mask** (`boolean numpy array`) - The signal locations marked as True are not used in the decomposition.
• **var_array** (`numpy array`) - Array of variance for the maximum likelihood PCA algorithm
• **var_func** (`function or numpy array`) - If function, it will apply it to the dataset to obtain the var_array. Alternatively, it can an array with the coefficients of a polynomial.
• **reproject** (`None | signal | navigation | both`) - 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 (the default is False) return any extra information if available

**Returns** `(X, E)` - If `algorithm` == `'RPCA_GoDec'` or `'ORPCA'` and `return_info` is True, returns the low-rank (X) and sparse (E) matrices from robust PCA.

**Return type** (numpy array, numpy array)

See also:
plot_decomposition_factors(), plot_decomposition_loadings(), plot_lev()

def get_bss_model(components=None)
Return the spectrum generated with the selected number of independent components

Parameters
components (`None, int, or list of ints`) - if None, rebuilds SI from all components if int, rebuilds SI from components in range 0-given int if list of ints, rebuilds SI from only components in given list

Returns

Return type  Signal instance

def get_decomposition_model(components=None)
Return the spectrum generated with the selected number of principal components

Parameters
components (`None, int, or list of ints`) - if None, rebuilds SI from all components if int, rebuilds SI from components in range 0-given int if list of ints, rebuilds SI from only components in given list

Returns

Return type  Signal instance
get_explained_variance_ratio()
Return the explained variation ratio of the PCA components as a Signal1D.

Returns

• \( s \) (Signal1D) – Explained variation ratio.

See Also

• plot_explained_variance_ratio, decomposition,
• get_decomposition_loadings,
• get_decomposition_factors.

normalize_bss_components(target='factors', function=<function sum>)
Normalize BSS components.

Parameters

• target ("factors", "loadings")–
• function (numpy universal function, optional, default np.sum)
  – Each target component is divided by the output of function(target). function must return a scalar when operating on numpy arrays and must have an axis.

normalize_decomposition_components(target='factors', function=<function sum>)
Normalize decomposition components.

Parameters

• target ("factors", "loadings")–
• function (numpy universal function, optional, default np.sum)
  – Each target component is divided by the output of function(target). function must return a scalar when operating on numpy arrays and must have an axis.

normalize_poissonian_noise(navigation_mask=None, signal_mask=None)
Scales the SI following Surf. Interface Anal. 2004; 36: 203–212 to “normalize” the poissonian data for decomposition analysis

Parameters

• navigation_mask (boolean numpy array)–
• signal_mask (boolean numpy array)–

plot_cumulative_explained_variance_ratio(n=50)
Plot the principal components explained variance up to the given number

Parameters n (int)–

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 (Scree Plot).

Parameters

• n (int or None) – Number of components to plot. If None, all components will be plot
• log (bool) – 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** *(True, False) : 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 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 or None)* – If None, a default figure will be created, otherwise will plot into fig

• **ax** *(matplotlib ax (subplot) or None)* – If None, a default ax will be created, otherwise will plot into ax

• ****kwargs** – remaining keyword arguments are passed to matplotlib.figure()

### Example

To generate a scree plot with customized symbols for signal vs. noise components and a modified cutoff threshold value:

```python
>>> 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'}
    noise_fmt={'marker': '*',
               's': 200,
               'c': 'green'})
```

Returns `ax`

Return type : `matplotlib.axes`
See also:

decomposition(), get_explained_variance_ratio(), get_decomposition_loadings(),
get_decomposition_factors()

reverse_bss_component(component_number)
Reverse the independent component

Parameters component_number (list or int) – component index/es

Examples

```python
>>> s = hs.load('some_file')
>>> s.decomposition(True)  # perform PCA
>>> s.blind_source_separation(3)  # perform ICA on 3 PCs
>>> s.reverse_bss_component(1)  # reverse IC 1
>>> s.reverse_bss_component((0, 2))  # reverse ICs 0 and 2
```

reverse_decomposition_component(component_number)
Reverse the decomposition component

Parameters component_number (list or int) – component index/es

Examples

```python
>>> s = hs.load('some_file')
>>> s.decomposition(True)  # perform PCA
>>> s.reverse_decomposition_component(1)  # reverse IC 1
>>> s.reverse_decomposition_component((0, 2))  # reverse ICs 0 and 2
```

undo_treatments()
Undo normalize_poissonian_noise

hyperspy.learn.mva.centering_and_whitening(X)

hyperspy.learn.mva.get_derivative(signal, diff_axes, diff_order)

hyperspy.learn.onmf module

class hyperspy.learn.onmf.ONMF (rank, lambda1=1.0, kappa=1.0, store_r=False, robust=False)

Bases: object

This class performs Online Robust NMF with missing or corrupted data.

fit ()
learn factors from the given data

project ()
project the learnt factors on the given data

finish ()
return the learnt factors and loadings
Notes

The ONMF code is based on a transcription of the OPGD algorithm MATLAB code obtained from the authors of the following research paper:


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).

Creates Online Robust NMF instance that can learn a representation

Parameters

- **rank** *(int)* – The rank of the representation (number of components/factors)
- **lambda1** *(float)* – Nuclear norm regularization parameter.
- **kappa** *(float)* – Sparse error regularization parameter.
- **store_r** *(bool)* – If True, stores the sparse error matrix, False by default.
- **robust** *(bool)* – If True, the original OPGD implementation is used for corruption/outlier regularization, otherwise L2-norm. False by default.

**finish()**
Return the learnt factors and loadings.

**fit**(X, batch_size=None)
Learn NMF components from the data.

Parameters

- **X** *(numpy.ndarray, iterator)* – [nsample x nfeatures] matrix of observations or an iterator that yields samples, each with nfeatures elements.
- **batch_size** *(None, int)* – If not None, learn the data in batches, each of batch_size samples or less.

**project**(X, return_R=False)
Project the learnt components on the data.

Parameters

- **X** *(numpy.ndarray, iterator)* – [nsample x nfeatures] matrix of observations or an iterator that yields samples, each with nfeatures elements.
- **return_R** *(bool)* – If True, returns the sparse error matrix as well. Otherwise only the weights (loadings)

**hyperspy.learn.onmf.onmf**(X, rank=1, lambda1=1, kappa=1, store_r=False, project=False, robust=False)

**hyperspy.learn.rpca module**

**class hyperspy.learn.rpca.ORPCA**(rank, fast=False, lambda1=None, lambda2=None, method=None, learning_rate=None, init=None, training_samples=None, momentum=None)

Bases: object

**finish()**

**fit**(X, iterating=None)
The ORPCA code is based on a transcription of MATLAB code obtained from the following research paper:


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 here:


This function performs Robust PCA with missing or corrupted data, using the GoDec algorithm.
Parameters

- **X** (numpy array) – is the [nfeatures x nsamples] matrix of observations.
- **rank** (int) – The model dimensionality.
- **lambda1** (None / float) – Regularization parameter. If None, set to 1 / sqrt(nsamples)
- **power** (None / integer) – The number of power iterations used in the initialization
  If None, set to 0 for speed
- **tol** (None / float) – Convergence tolerance If None, set to 1e-3
- **maxiter** (None / integer) – Maximum number of iterations If None, set to 1e3

Returns

- **Xhat** (numpy array) – is the [nfeatures x nsamples] low-rank matrix
- **Ehat** (numpy array) – is the [nfeatures x nsamples] sparse error matrix
- **Ghat** (numpy array) – is the [nfeatures x nsamples] Gaussian noise matrix
- **U, S, V** (numpy arrays) – are the results of an SVD on Xhat

Notes


Code: https://sites.google.com/site/godecomposition/matrix/artifact-1

**hyperspy.learn.svd_pca module**

**hyperspy.learn.svd_pca.svd_pca**(data, fast=False, output_dimension=None, centre=None, auto_transpose=True)

Perform PCA using SVD.

Parameters

- **data** (numpy array) – MxN array of input data (M variables, N trials)
- **fast** (bool) – Wheter to use randomized svd estimation to estimate a limited number of componentes given by output_dimension
- **output_dimension** (int) – Number of components to estimate when fast is True
- **centre** (None / 'variables' / 'trials') – If None no centring is applied. If ‘variable’ the centring will be performed in the variable axis. If ‘trials’, the centring will be performed in the ‘trials’ axis.
- **auto_transpose** (bool) – If True, automatically transposes the data to boost performance

Returns

- **factors** (numpy array)
- **loadings** (numpy array)
- **explained_variance** (numpy array)
- **mean** (numpy array or None (if center is None))
hyperSpy Documentation, Release 1.5.1.dev

Module contents

hyperspy.miss package

Subpackages

hyperspy.misc.eds package

Submodules

hyperspy.misc.eds.ffast_mac module

hyperspy.misc.eds.utils module

hyperspy.misc.eds.utils.edx_cross_section_to_zeta(cross_sections, elements)

Convert a list of cross_sections in barns (b) to zeta-factors (kg/m^2).

Parameters

- cross_section (list of float) – A list of cross sections in barns.
- elements (list of str) – A list of element chemical symbols in the same order as
  the cross sections e.g. ['Al', 'Zn']

Returns zeta_factors – zeta_factors with units kg/m^2.

Return type list of float

hyperspy.misc.eds.utils.electron_range(element, beam_energy, density='auto', tilt=0)

Returns the maximum electron range for a pure bulk material according to the Kanaya-Okayama parameterization.

Parameters

- element (str) – The element symbol, e.g. 'Al'.
- beam_energy (float) – The energy of the beam in keV.
- density (float, 'auto') – The density of the material in g/cm^3. If 'auto', the
density of the pure element is used.
- tilt (float) – The tilt of the sample in degrees.

Returns

Return type Electron range in micrometers.

Examples

```python
>>> # Electron range in pure Copper at 30 kV in micron
>>> hs.eds.electron_range('Cu', 30.)
2.8766744984001607
```
Notes

See also the textbook of Goldstein et al., Plenum publisher, third edition p 72.

`hyperspy.misc.eds.utils.get_FWHM_at_Energy`(`energy_resolution_MnKa`, `E`)
Calculates an approximate FWHM, accounting for peak broadening due to the detector, for a peak at energy `E` given a known width at a reference energy.

The factor 2.5 is a constant derived by Fiori & Newbury as references below.

Parameters

• `energy_resolution_MnKa` (`float`) – Energy resolution of Mn Ka in eV
• `E` (`float`) – Energy of the peak in keV

Returns `float`
Return type FWHM of the peak in keV

Notes

This method implements the equation derived by Fiori and Newbury as is documented in the following:


`hyperspy.misc.eds.utils.get_xray_lines_near_energy`(`energy`, `width=0.2`, `only_lines=None`)
Find xray lines near a specific energy, more specifically all xray lines that satisfy `only_lines` and are within the given energy window width around the passed energy.

Parameters

• `energy` (`float`) – Energy to search near in keV
• `width` (`float`) – Window width in keV around energy in which to find nearby energies, i.e. a value of 0.2 keV (the default) means to search +/- 0.1 keV.
• `only_lines` – If not None, only the given lines will be added (eg. (‘a’,’Kb’)).

Returns

Return type List of xray-lines sorted by energy difference to given energy.

`hyperspy.misc.eds.utils.quantification_cliff_lorimer`(`intensities`, `kfactors`, `mask=None`)
Quantification using Cliff-Lorimer

Parameters

• `intensities` (`numpy.array`) – the intensities for each X-ray lines. The first axis should be the elements axis.
• `kfactors` (`list of float`) – The list of kfactor in same order as intensities eg. kfactors = [1.1.47.1.72] for [‘Al_Ka’,’Cr_Ka’, ‘Ni_Ka’]
• `mask` (`array of bool`) – The mask with the dimension of intensities[0]. If a pixel is True, the composition is set to zero.

Returns

3.1. hyperspy package
hyperSpy Documentation, Release 1.5.1.dev

- numpy.array containing the weight fraction with the same
  shape as intensities.

hyperspy.misc.eds.utils.quantification_cross_section(intensities, cross_sections, dose)

Quantification using EDX cross sections Calculate the atomic composition and the number of atoms per pixel from the raw X-ray intensity:

:param intensity: The integrated intensity for each X-ray line, where the first axis is the element axis.

Parameters

- cross_sections (list of floats) – List of X-ray scattering cross-sections in the same order as the intensities.
- dose (float) – the dose per unit area given by \( i \cdot t \cdot N / A \), \( i \) the current, \( t \) the acquisition time, and \( N \) the number of electron by unit electric charge.

Returns

- numpy.array containing the atomic fraction of each element, with the same shape as the intensity input.
- numpy.array of the number of atoms counts for each element, with the same shape as the intensity input.

hyperspy.misc.eds.utils.quantification_zeta_factor(intensities, zfactors, dose)

Quantification using the zeta-factor method

Parameters

- intensities (numpy.array) – The intensities for each X-ray line. The first axis should be the elements axis.
- zfactors (list of float) – The list of zeta-factors in the same order as intensities e.g. zfactors = [628.10, 539.89] for ['As_Ka', 'Ga_Ka'].
- dose (float) – The total electron dose given by \( i \cdot t \cdot N \), \( i \) the current, \( t \) the acquisition time and \( N \) the number of electrons per unit electric charge (1/e).

Returns

- A numpy.array containing the weight fraction with the same shape as intensities and mass thickness in kg/m².

hyperspy.misc.eds.utils.take_off_angle(tilt_stage, azimuth_angle, elevation_angle)

Calculate the take-off-angle (TOA).

TOA is the angle with which the X-rays leave the surface towards the detector.

Parameters

- tilt_stage (float) – The tilt of the stage in degrees. The sample is facing the detector when positively tilted.
- azimuth_angle (float) – The azimuth of the detector in degrees. 0 is perpendicular to the tilt axis.
- elevation_angle (float) – The elevation of the detector in degrees.

Returns take_off_angle – In degrees.

Return type float.
Examples

```python
>>> hs.eds.take_off_angle(tilt_stage=10.,
                      azimuth_angle=45., elevation_angle=22.)
28.865971201155283
```

Notes

Defined by M. Schaffer et al., Ultramicroscopy 107(8), pp 587-597 (2007)

```python
hyperspy.misc.eds.utils.xray_lines_model(elements, beam_energy=200,
                                          weight_percents=None,
                                          energy_resolution_MnKa=130,
                                          energy_axis=None)
```

Generate a model of X-ray lines using a Gaussian distribution for each peak.

The area under a main peak (alpha) is equal to 1 and weighted by the composition.

Parameters

- **elements** (list of strings) – A list of chemical element symbols.
- **beam_energy** (float) – The energy of the beam in keV.
- **weight_percents** (list of float) – The composition in weight percent.
- **energy_resolution_MnKa** (float) – The energy resolution of the detector in eV.
- **energy_axis** (dict) – The dictionary for the energy axis. It must contains ‘size’ and the units must be ‘eV’ of ‘keV’.

Example

```python
>>> s = utils_eds.simulate_model(['Cu', 'Fe'], beam_energy=30)
>>> s.plot()
```

```python
hyperspy.misc.eds.utils.xray_range(xray_line, beam_energy, density='auto')
```

Return the maximum range of X-ray generation according to the Anderson-Hasler parameterization.

Parameters

- **xray_line** (str) – The X-ray line, e.g. ‘Al_Ka’
- **beam_energy** (float) – The energy of the beam in kV.
- **density** (float, 'auto') – The density of the material in g/cm3. If ‘auto’, the density of the pure element is used.

Returns

Return type X-ray range in micrometer.

Examples

```python
>>> # X-ray range of Cu Ka in pure Copper at 30 kV in micron
>>> hs.eds.xray_range('Cu_Ka', 30.)
1.9361716759499248
```
>>> # X-ray range of Cu Ka in pure Carbon at 30kV in micron
>>> hs.eds.xray_range('Cu_Ka', 30., hs.material.elements.C.
>>> Physical_properties.density_gcm3)
7.6418811280855454

Notes


See also the textbook of Goldstein et al., Plenum publisher, third edition p 286

hyperspy.misc.eds.utils.zeta_to_edx_cross_section(zfactors, elements)
Convert a list of zeta-factors (kg/m^2) to cross_sections in barns (b).

Parameters

- **zfactors** *(list of float)* – A list of zeta-factors.
- **elements** *(list of str)* – A list of element chemical symbols in the same order as the cross sections e.g. ['Al', 'Zn']

Returns **cross_sections** – cross_sections with units in barns.

Return type  list of float

Module contents

hyperspy.misc.eels package

Submodules

hyperspy.misc.eels.base_gos module

class hyperspy.misc.eels.base_gos.GOSBase
    Bases: object
    get_parametrized_energy_axis(*kl, *n)
    get_parametrized_qaxis(*kl, *n)
    get_qaxis_and_gos(ienergy, qmin, qmax)
    read_elements()

hyperspy.misc.eels.eelsdb module

This module provides tools to interact with The EELS Database.

hyperspy.misc.eels.eelsdb.eelsdb(spectrum_type=None, title=None, author=None, element=None, formula=None, edge=None, min_energy=None, max_energy=None, resolution=None, min_energy_compare='gt', max_energy_compare='lt', resolution_compare='lt', max_n=-1, monochromated=None, order=None, order_direction='ASC')

Download spectra from the EELS Data Base.
Parameters

- **spectrum_type** ([{'coreloss', 'lowloss', 'zeroloss', 'xrayabs'}, optional])

- **title** (string) – Search spectra titles for a text string.

- **author** (string, optional) – Search authors for a text string.

- **element** (string or list of strings, optional) – Filter for the presence of one or more elements. Each string must correspond with a valid element symbol.

- **formula** (string) – Chemical formula of the sample.

- **edge** ([K, L1, L2, 3', M2, 3', M4, 5', N2, 3', N4, 5', O2, 3', O4, 5'], optional) – Filter for spectra with a specific class of edge.

- **max_energy** (min_energy,) – Minimum and maximum energy in eV.

- **resolution** (float, optional) – Energy resolution in eV.

- **resolution_compare** (["lt", "eq", "gt"], optional, default "lt") – “lt” to search for all spectra with resolution less than resolution. “eq” for equal, “gt” for greater than.

- **max_energy_compare** (min_energy_compare,) – “lt” to search for all spectra with min/max energy less than min_energy’max_energy’. “eq” for equal, “gt” for greater than. Default values are “gt”’lt” for min_energy’max_energy’ respectively.

- **monochromated** (bool or None (default)) –

- **max_n** (int, default -1) – Maximum number of spectra to return. -1 to return all.


- **order_direction** (["ASC", "DESC"]) – Sorting order direction.

Returns spectra – A list containing all the spectra matching the given criteria if any.

Return type list

**hyperspy.misc.eels.effective_angle module**

**hyperspy.misc.eels.effective_angle.effective_angle** (E0, E, alpha, beta)

Calculates the effective collection angle

Parameters

- **E0** (float) – electron beam energy in keV

- **E** (float) – energy loss in eV

- **alpha** (float) – convergence angle in mrad
• **beta** (*float*) – collection angle in mrad

**Returns** *float*

**Return type** effective collection angle in mrad

**Notes**

Code translated to Python from Egerton (second edition) page 420

### hyperspy.misc.eels.hartree_slater_gos module

**class** `hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS(element_subshell)`

**Bases:** `hyperspy.misc.eels.base_gos.GOSBase`

Read Hartree-Slater Generalized Oscillator Strength parametrized from files.

**Parameters**

`element_subshell` (*str, dict*) – Usually a string, for example, ‘Ti_L3’ for the GOS of the titanium L3 subshell. If a dictionary is passed, it is assumed that Hartree Slater GOS was exported using `GOS.as_dictionary`, and will be reconstructed.

`readgosfile()`

Read the GOS files of the element subshell from the location defined in Preferences.

`get_qaxis_and_gos(ienergy, qmin, qmax)`

given the energy axis index and qmin and qmax values returns the qaxis and gos between qmin and qmax using linear interpolation to include qmin and qmax in the range.

`as_dictionary()`

Export the GOS as a dictionary that can be saved.

**energy_axis**

The tabulated energy axis

**Type** array

**qaxis**

The tabulated qaxis

**Type** array

**energy_onset**

The energy onset for the given element subshell as obtained from internal tables.

**Type** *float*

**Parameters**

`element_subshell` (*str*) – For example, ‘Ti_L3’ for the GOS of the titanium L3 subshell

`as_dictionary(fullcopy=True)`

Export the GOS as a dictionary

`integrateq(onset_energy, angle, E0)`

`readgosfile()`
class hyperspy.misc.eels.hydrogenic_gos.HydrogenicGOS(element_subshell)

Bases: hyperspy.misc.eels.base_gos.GOSBase

Computes the K and L GOS using R. Egerton’s routines.

Parameters

  element_subshell (str) – For example, ‘Ti_L3’ for the GOS of the titanium L3 subshell

parametrize_GOS()

  Parametrize the GOS to speed up the calculation.

get_qaxis_and_gos(energy, qmin, qmax)

  Given the energy axis index and qmin and qmax values returns the qaxis and gos between qmin and qmax using linear interpolation to include qmin and qmax in the range.

energy_axis

  The tabulated energy axis

    Type array

qaxis

  The tabulated qaxis

    Type array

energy_onset

  The energy onset for the given element subshell as obtained from internal tables.

    Type float

Notes

The Hydrogenic GOS are calculated using R. Egerton’s SIGMAK3 and SIGMAL3 routines that has been translated from Matlab to Python by I. Iyengar. See http://www.tem-eels.ca/ for the original code.

Parameters

  element_subshell (str) – For example, ‘Ti_L3’ for the GOS of the titanium L3 subshell

gosfuncK(E, qa02)
gosfuncL(E, qa02)

integrateq(onset_energy, angle, E0)

hyperspy.misc.eels.tools module

def hyperspy.misc.eels.tools.eels_constant(s, zlp, t)

Calculate the constant of proportionality (k) in the relationship between the EELS signal and the dielectric function. dielectric function from a single scattering distribution (SSD) using the Kramers-Kronig relations.

\[
S(E) = \frac{I_0t}{\pi a_0 m_0 v^2} \ln \left[ 1 + \left( \frac{\beta}{\bar{\theta}_E} \right)^2 \right] \Im\left( \frac{-1}{\epsilon(E)} \right) = k \Im\left( \frac{-1}{\epsilon(E)} \right)
\]

Parameters
• \textbf{zlp}((number, BaseSignal)) – If the ZLP is the same for all spectra, the integral of the ZLP can be provided as a number. Otherwise, if the ZLP intensity is not the same for all spectra, it can be provided as i) a Signal of the same dimensions as the current signal containing the ZLP spectra for each location ii) a Signal of signal dimension 0 and navigation\_dimension equal to the current signal containing the integrated ZLP intensity.

• \textbf{t}((None, number, BaseSignal)) – The sample thickness in nm. If the thickness is the same for all spectra it can be given by a number. Otherwise, it can be provided as a Signal with signal dimension 0 and navigation\_dimension equal to the current signal.

\textbf{Returns k}

\textbf{Return type} Signal instance

\textit{hyperSpy.misc.eels.tools.estimate_variance_parameters(noisy\_signal, clean\_signal, mask=None, pol\_order=1, higher\_than=None, return\_results=False, plot\_results=True, weighted=False, store\_results='ask')}

Find the scale and offset of the Poissonian noise

By comparing an SI with its denoised version (i.e. by PCA), this plots an estimation of the variance as a function of the number of counts and fits a polynomy to the result.

\textbf{Parameters}

• \textbf{clean\_SI (noisy\_SI,)} –

• \textbf{mask (numpy bool array)} – To define the channels that will be used in the calculation.

• \textbf{pol\_order (int)} – The order of the polynomy.

• \textbf{higher\_than (float)} – To restrict the fit to counts over the given value.

• \textbf{return\_results (Bool)} –

• \textbf{plot\_results (Bool)} –

• \textbf{store\_results ((True, False, "ask"), default "ask")} – If True, it stores the result in the signal metadata

\textbf{Returns}

• Dictionary with the result of a linear fit to estimate the offset

• and scale factor

\textit{hyperSpy.misc.eels.tools.power\_law\_perc\_area(E1, E2, r)}

\textit{hyperSpy.misc.eels.tools.ratio(edge\_A, edge\_B)}

\textit{hyperSpy.misc.eels.tools.rel\_std\_of\_fraction(a, std\_a, b, std\_b, corr\_factor=1)}

\textbf{Module contents}

\textit{hyperSpy.misc.holography package}

\textbf{Submodules}
hyperspy.misc.holography.reconstruct module

hyperspy.misc.holography.reconstruct.aperture_function(r, apradius, rsmooth)
A smooth aperture function that decays from apradius-rsmooth to apradius+rsmooth.

Parameters

• r (ndarray) – Array of input data (e.g. frequencies)
• apradius (float) – Radius (center) of the smooth aperture. Decay starts at apradius - rsmooth.
• rsmooth (float) – Smoothness in halfwidth. rsmooth = 1 will cause a decay from 1 to 0 over 2 pixel.

hyperspy.misc.holography.reconstruct.estimate_sideband_position(holo_data, holo_sampling, central_band_mask_radius=None, sb='lower', high_cf=True)

Finds the position of the sideband and returns its position.

Parameters

• holo_data (ndarray) – The data of the hologram.
• holo_sampling (tuple) – The sampling rate in both image directions.
• central_band_mask_radius (float, optional) – The aperture radius used to mask out the centerband.
• sb (str, optional) – Chooses which sideband is taken. ‘lower’, ‘upper’, ‘left’, or ‘right’.
• high_cf (bool, optional) – If False, the highest carrier frequency allowed for the sideband location is equal to half of the Nyquist frequency (Default: True).

Returns

Return type Tuple of the sideband position (y, x), referred to the unshifted FFT.

hyperspy.misc.holography.reconstruct.estimate_sideband_size(sb_position, holo_shape, sb_size_ratio=0.5)

Estimates the size of sideband filter

Parameters

• holo_shape (array_like) – Holographic data array
• sb_position (tuple) – The sideband position (y, x), referred to the non-shifted FFT.
• sb_size_ratio (float, optional) – Size of sideband as a fraction of the distance to central band

Returns sb_size – Size of sideband filter

Return type float

hyperspy.misc.holography.reconstruct.freq_array(shape, sampling)

Makes up a frequency array.

Parameters
- **shape (tuple)** – The shape of the array.
- **sampling (tuple)** – The sampling rates of the array.

**Returns**

**Return type** Array of the frequencies.

```python
def reconstruct(holo_data, holo_sampling, sb_size, sb_position, sb_smoothness, output_shape=None, plotting=False)
```

Core function for holographic reconstruction.

**Parameters**

- **holo_data (array_like)** – Holographic data array
- **holo_sampling (tuple)** – Sampling rate of the hologram in y and x direction.
- **sb_size (float)** – Size of the sideband filter in pixel.
- **sb_position (tuple)** – Sideband position in pixel.
- **sb_smoothness (float)** – Smoothness of the aperture in pixel.
- **output_shape (tuple, optional)** – New output shape.
- **plotting (boolean)** – Plots the masked sideband used for reconstruction.

**Returns**

**wav** – Reconstructed electron wave

**Return type** nparray

**hyperspy.misc.holography.tools module**

```python
def calculate_carrier_frequency(holo_data, sb_position, scale)
```

Calculates fringe carrier frequency of a hologram

**Parameters**

- **holo_data (ndarray)** – The data of the hologram.
- **sb_position (tuple)** – Position of the sideband with the reference to non-shifted FFT
- **scale (tuple)** – Scale of the axes that will be used for the calculation.

**Returns**

**Return type** Carrier frequency

```python
def estimate_fringe_contrast_fourier(holo_data, sb_position, apodization='hanning')
```

Estimates average fringe contrast of a hologram by dividing amplitude of maximum pixel of sideband by amplitude of FFT’s origin.

**Parameters**

- **holo_data (ndarray)** – The data of the hologram.
- **sb_position (tuple)** – Position of the sideband with the reference to non-shifted FFT
• **apodization** *(string, None)* – Use ‘hanning’, ‘hamming’ or None to apply apodization window in real space before FFT. Apodization is typically needed to suppress the striking due to sharp edges of the which often results in underestimation of the fringe contrast. (Default: ‘hanning’)

**Returns**

**Return type** Fringe contrast as a float

**Module contents**

**hyperspy.misc.io package**

**Submodules**

**hyperspy.misc.io.fei_stream_readers module**

```python
class hyperspy.misc.io.fei_stream_readers.DenseSliceCOO(coords, data=None, shape=None, has_duplicates=True, sorted=False, prune=False, cache=False, fill_value=None)
```

*Bases: sparse.coo.core.COO*

Just like sparse.COO, but returning a dense array on indexing/slicing

**hyperspy.misc.io.fei_stream_readers.array_to_stream**

Convert an array to a FEI stream

**Parameters**

array *(array)* –

**hyperspy.misc.io.fei_stream_readers.stream_to_array** *(stream, spatial_shape, channels, first_frame=0, rebin_energy=1, sum_frames=True, dtype='uint16', spectrum_image=None)*

Returns data stored in a FEI stream as a nd COO array

**Parameters**

- **stream** *(numpy array)* –
- **spatial_shape** *(tuple of ints)* – (ysize, xsize)
- **channels** *(ints)* – Number of channels in the spectrum
- **rebin_energy** *(int)* – Rebin the spectra. The default is 1 (no rebinning applied)
- **sum_frames** *(bool)* – If True, sum all the frames
- **dtype** *(numpy dtype)* – dtype of the array where to store the data
- **number_of_frame** *(int or None)* –
- **spectrum_image** *(numpy array or None)* – If not None, the array provided will be filled with the data in the stream.
hyperspy.misc.io.fei_stream_readers.stream_to_sparse_COO_array(stream_data, spatial_shape, channels, last_frame, rebin_energy=1, sum_frames=True, first_frame=0)

Returns data stored in a FEI stream as a nd COO array

Parameters

- **stream_data** (numpy array) –
- **spatial_shape** (tuple of ints) – (ysize, xsize)
- **channels** (ints) – Number of channels in the spectrum
- **rebin_energy** (int) – Rebin the spectra. The default is 1 (no rebinning applied)
- **sum_frames** (bool) – If True, sum all the frames

hyperspy.misc.io.tools module

hyperspy.misc.io.tools.append2pathname(filename, to_append)

Append a string to a path name

Parameters

- **filename** (str) –
- **to_append** (str) –

hyperspy.misc.io.tools.dump_dictionary(file, dic, string='root', node_separator='.', value_separator=' = ')

hyperspy.misc.io.tools.ensure_directory(path)

Check if the path exists and if it does not create the directory

hyperspy.misc.io.tools.incremental_filename(filename, i=1)

If a file with the same file name exists, returns a new filename that does not exists.

The new file name is created by appending `-n` (where `n` is an integer) to path name

Parameters

- **filename** (str) –
- **i** (int) – The number to be appended.

hyperspy.misc.io.tools.overwrite(fname)

If file exists ‘fname’, ask for overwriting and return True or False, else return True.

hyperspy.misc.io.utils_readfile module

hyperspy.misc.io.utils_readfile.read_boolean(f, endian)

Read a 1-Byte character from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_byte(f, endian)

Read a 1-Byte character from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_char(f, endian)

Read a 1-Byte character from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.
hyperspy.misc.io.utils_readfile.read_double(f, endian)
Read a 8-Byte floating point from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_float(f, endian)
Read a 4-Byte floating point from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_long(f, endian)
Read a 4-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_long_long(f, endian)
Read a 8-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_short(f, endian)
Read a 2-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy.misc.io.utils_readfile.read_ulong(f, endian)
Read a 4-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy/misc/io/utils_readfile.read_ulong_long(f, endian)
Read a 8-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

hyperspy/misc/io/utils_readfile.read_ushort(f, endian)
Read a 2-Byte integer from file f with a given endianness (byte order). endian can be either ‘big’ or ‘little’.

Module contents

hyperspy.misc.machine_learning package

Submodules

hyperspy/misc/machine_learning/import_sklearn module

Import sklearn, fast_svd and randomized_svd from scikits-learn with support for multiple versions

hyperspy/misc/machine_learning/orthomax module

hyperspy.misc.machine_learning.orthomax.orthomax(A, gamma=1, reltol=1.4901e-07, maxit=256)

hyperspy/misc/machine_learning/tools module

hyperspy/misc/machine_learning/tools.amari(C, A)
Amari test for ICA Adapted from the MILCA package http://www.klab.caltech.edu/~kraskov/MILCA/

Parameters

- C (numpy array) –
- A (numpy array) –
hyperspy.misc.array_tools module

hyperspy.misc.array_tools.are_aligned(shape1, shape2)

Check if two numpy arrays are aligned.

Parameters

- shape2 (shape1) –

Returns

isaligned

Return type

bool

dict2sarray(dictionary, sarray=None, dtype=None)

Populates a struct array from a dictionary

Parameters

- dictionary (dict) –

- sarray (struct array or None) – Either sarray or dtype must be given. If sarray is given, it is populated from the dictionary.

- dtype (None, numpy dtype or dtype list) – If sarray is None, dtype must be given. If so, a new struct array is created according to the dtype, which is then populated.

Returns

Return type

Structure array

get_array_memory_size_in_GiB(shape, dtype)

Given the size and dtype returns the amount of memory that such an array needs to allocate

Parameters

- shape (tuple) –

- dtype (data-type) – The desired data-type for the array.

homogenize_ndim(*args)

Given any number of arrays returns the same arrays reshaped by adding facing dimensions of size 1.

numba_histogram

param data: Input data. The histogram is computed over the flattened array. :type data: numpy array

param bins: Number of bins :type bins: int :param ranges: The lower and upper range of the bins. :type ranges: (float, float)

Returns

hist – The values of the histogram.

Return type

array

rebin(a, new_shape=None, scale=None, crop=True)

Rebin array.

rebin ndarray data into a smaller or larger array based on a linear interpolation. Specify either a new_shape or a scale. Scale of 1== no binning. Scale less than one results in up-sampling.

Parameters
• a (numpy array) –
• new_shape (a list of floats or integer, default None) – For each dimension specify the new_shape of the np.array. This will then be converted into a scale.
• scale (a list of floats or integer, default 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 numpy array. *Note: Only one of scale or new_shape should be specified otherwise the function will not run*
• crop (bool, default True) – When binning by a non-integer number of pixels it is likely that the final row in each dimension contains less than the full quota to fill one pixel. e.g. 5*5 array binned by 2.1 will produce two rows containing 2.1 pixels and one row containing only 0.8 pixels worth. 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.

Returns

Return type  numpy array

Examples

```python
>>> a=rand(6,4); b=rebin(a,scale=(3,2))
>>> a=rand(6); b=rebin(a,scale=(2,))
```

Notes

Fast re_bin function Adapted from scipy cookbook If rebin function fails with error stating that the function is ‘not binned and therefore cannot be rebinned’, add binned to metadata with: >>> s.metadata.Signal.binned = True

```python
hyperspy.misc.array_tools.sarray2dict (sarray, dictionary=None)
```

Converts a struct array to an ordered dictionary

Parameters

• sarray (struct array) –
• dictionary (None or dict) – If dictionary is not None the content of sarray will be appended to the given dictionary

Returns

Return type  Ordered dictionary

```python
hyperspy.misc.config_dir module

hyperspy.misc.date_time_tools module

hyperspy.misc.date_time_tools.ISO_format_to_serial_date (date, time, timezone='UTC')
```

Convert ISO format to a serial date.
hyperSpy Documentation, Release 1.5.1.dev

hyperSpy.misc.date_time_tools.datetime_to_serial_date(dt)
Convert datetime.datetime object to a serial date.

hyperSpy.misc.date_time_tools.get_date_time_from_metadata(metadata, formatting='ISO')
Get the date and time from a metadata tree.

Parameters

- metadata (metadata object) -
- formatting (string, ('ISO', 'datetime', 'datetime64')) - Default: 'ISO'. This parameter set the formatting of the date, and the time, it can be ISO 8601 string, datetime.datetime or a numpy.datetime64 object. In the later case, the time zone is not supported.

Returns

Return type  string, datetime.datetime or numpy.datetime64 object

Example

```python
>>> s = hs.load("example1.msa")
>>> s.metadata
<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>date = 1991-10-01</td>
</tr>
<tr>
<td>original_filename = example1.msa</td>
</tr>
<tr>
<td>time = 12:00:00</td>
</tr>
<tr>
<td>title = NIO EELS OK SHELL</td>
</tr>
</tbody>
</table>

>>> s = get_date_time_from_metadata(s.metadata)
'1991-10-01T12:00:00'
>>> s = get_date_time_from_metadata(s.metadata, formatting='ISO')
'1991-10-01T12:00:00'
>>> s = get_date_time_from_metadata(s.metadata, formatting='datetime')
>>> s = get_date_time_from_metadata(s.metadata, formatting='datetime64')
```

hyperSpy.misc.date_time_tools.serial_date_to_ISO_format(serial)
Convert serial_date to a tuple of string (date, time, time_zone) in ISO format. By default, the serial date is converted in local time zone.

hyperSpy.misc.date_time_tools.serial_date_to_datetime(serial)
Convert serial date to a datetime.datetime object.

hyperSpy.misc.date_time_tools.update_date_time_in_metadata(dt, metadata)
Update the date and time in a metadata tree.

Parameters

- dt (date and time information: it can be a ISO 8601 string,) - a datetime.datetime or a numpy.datetime64 object
- metadata (metadata object to update) -

Returns

Return type  metadata object
Example

```python
>>> s = hs.load("example1.msa")
>>> dt = '2016-12-12T12:12:12-05:00'
>>> s.metadata = update_date_time_in_metadata(dt, s.metadata)
>>> s.metadata
General
  date = 2016-12-12
  original_filename = example1.msa
  time = 12:12:12
  time_zone = 'EST'
  title = NIO EELS OK SHELL
```

**hyperspy.misc.elements module**

**hyperspy.misc.example_signals_loading module**

**hyperspy.misc.example_signals_loading.load_1D_EDS_SEM_spectrum()**

Load an EDS-SEM spectrum

**Notes**

- Sample: EDS-TM002 provided by BAM (www.webshop.bam.de)
- SEM Microscope: Nvision40 Carl Zeiss
- EDS Detector: X-max 80 from Oxford Instrument

**hyperspy.misc.example_signals_loading.load_1D_EDS_TEM_spectrum()**

Load an EDS-TEM spectrum

**Notes**

- Sample: FePt bimetallic nanoparticles
- SEM Microscope: Tecnai Osiris 200 kV D658 AnalyticalTwin
- EDS Detector: Super-X 4 detectors Brucker

**hyperspy.misc.example_signals_loading.load_object_hologram()**

Load an object hologram image

**Notes**

- Sample: Fe needle with YOx nanoparticle inclusions [Migunov, V. et al.

HyperSpy Documentation, Release 1.5.1.dev

hyperspy.misc.example_signals_loading.load_reference_hologram()
Load a reference hologram image

Notes

• Sample: Fe needle with YOx nanoparticle inclusions [Migunov, V. et al.

hyperspy.misc.export_dictionary module

hyperspy.misc.export_dictionary.check_that_flags_make_sense(flags)

hyperspy.misc.export_dictionary.export_to_dictionary(target, whitelist, dic, full-copy=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 (dictionary) – 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 dill 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

• dic (dictionary) – A dictionary where the object will be exported
• fullcopy (bool) – Copies of objects are stored, not references. If any found, functions
  will be pickled and signals converted to dictionaries

hyperspy.misc.export_dictionary.load_from_dictionary(target, dic)
Loads attributes of target to dictionary dic The attribute list is read from dic[‘whitelist’].keys()

Parameters

• target (object) – must contain the (nested) attributes of the whitelist.keys()
• **dic (dictionary)** – A dictionary, containing field `_whitelist`, which is a dictionary with all keys that were exported, with values being flag strings. The convention of the flags is as follows:
  * `init`: object used for initialization of the target. Will be copied to the `_whitelist` after loading
  * `fn`: the targeted attribute is a function, and may have been pickled (preferably with dill package).
  * `id`: the id of the original object was exported and the attribute will not be set. The key has to be `'_id_'`
  * `sig`: The targeted attribute was a signal, and may have been converted to a dictionary if `fullcopy=True`

```python
def parse_flag_string(flags):
    # Implementation of parse_flag_string

def reconstruct_object(flags, value):
    # Implementation of reconstruct_object
```

**hyperspy.misc.ipython_tools module**

```python
from hyperspy.misc.ipython_tools import get_interactive_ns, get_ipython,
                   is_it_running_from_ipython, turn_logging_off,
                   turn_logging_on
```

**hyperspy.misc.material module**

```python
from hyperspy.misc.material import atomic_to_weight
```

```python
def atomic_to_weight(atomic_percent, elements='auto'):
    # Implementation of atomic_to_weight
```

**Examples**

Calculate the weight percent of modern bronze given its atomic percent: >>>

```python
hs.material.atomic_to_weight([93.2, 6.8], ('Cu', 'Sn'))
```

Returns weight_percent – composition in weight percent.

**Return type** as atomic_percent
hyperspy.misc.material.density_of_mixture(weight_percent, elements='auto', mean='harmonic')

Calculate the density of a mixture of elements.

The density of the elements is retrieved from an internal database. The calculation is only valid if there is no interaction between the components.

Parameters

- **weight_percent** (list of float or list of signals) – A list of weight percent for the different elements. If the total is not equal to 100, each weight percent is divided by the sum of the list (normalization).

- **elements** (list of str) – A list of element symbols, e.g. ['Al', 'Zn']. If elements is 'auto', take the elements in each signal metadata of the weight_percent list.

- **mean** ('harmonic' or 'weighted') – The type of mean use to estimate the density

Returns: density

Return type: The density in g/cm³.

Examples

Calculate the density of modern bronze given its weight percent: >>> hs.material.density_of_mixture([88, 12], ['Cu', 'Sn'])

hyperspy.misc.material.mass_absorption_coefficient(element, energies)

Mass absorption coefficient (\(\mu/\rho\)) of a X-ray absorbed in a pure material.

The mass absorption is retrieved from the database of Chantler2005

Parameters

- **element** (str) – The element symbol of the absorber, e.g. ‘Al’.

- **energies** (float or list of float or str or list of str) – The energy or energies of the X-ray in keV, or the name of the X-rays, e.g. ‘Al_Ka’.

Returns

Return type: mass absorption coefficient(s) in cm²/g

Examples

>>> hs.material.mass_absorption_coefficient(
    >>>    element='Al', energies=['C_Ka','Al_Ka'])

array([ 26330.38933818, 372.02616732])

See also:

hs.material.mass_absorption_mixture()

The mass absorption coefficient is calculated as a weighted mean of the weight percent and is retrieved from the database of Chantler2005.

**Parameters**

- **weight_percent (list of float or list of signals)** – The composition of the absorber(s) in weight percent. The first dimension of the matrix corresponds to the elements.
- **elements (list of str or 'auto')** – The list of element symbol of the absorber, e.g. ['Al','Zn']. If elements is 'auto', take the elements in each signal metadata of the weight_percent list.
- **energies (list of float or list of str or 'auto')** – The energy or energies of the X-ray in keV, or the name of the X-rays, e.g. ‘Al_Ka’. If ‘auto’, take the lines in each signal metadata of the weight_percent list.

**Examples**

```python
>>> hs.material.mass_absorption_mixture(
    elements=['Al','Zn'], weight_percent=[50,50], energies='Al_Ka')
2587.41616439
```

**Returns**

- float or array of float
- mass absorption coefficient(s) in cm²/g

**See also:**

hs.material.mass_absorption_coefficient()


**hyperspy.misc.material.weight_to_atomic (weight_percent, elements='auto')**

Convert weight percent (wt%) to atomic percent (at.%).

**Parameters**

- **weight_percent (list of float or list of signals)** – The weight fractions (composition) of the sample.
- **elements (list of str)** – A list of element abbreviations, e.g. ['Al','Zn']. If elements is ‘auto’, take the elements in each signal metadata of the weight_percent list.

**Returns** atomic_percent – Composition in atomic percent.

**Return type** as weight_percent

**Examples**

Calculate the atomic percent of modern bronze given its weight percent: >>> hs.material.weight_to_atomic((88, 12), ('Cu', 'Sn')) array([ 93.19698614, 6.80301386])

3.1. hyperspy package
**hyperspy.misc.math_tools module**

- `hyperspy.misc.math_tools.antisymmetrize(a)`: Antisymmetrize an array.
- `hyperspy.misc.math_tools.anyfloatin(things)`: Check if iterable contains any non integer.
- `hyperspy.misc.math_tools.closest_nice_number(number)`: Get the closest nice number.
- `hyperspy.misc.math_tools.get_linear_interpolation(p1, p2, x)`: Get linear interpolation.
  - Parameters:
    - `p1, p2 ((x, y))`: Points.
    - `x (float)`: X value.
  - Returns: `y` for a given `x` for `y = ax + b`.
  - Return type: `float`.
- `hyperspy.misc.math_tools.hann_window_nth_order(m, order)`: Calculate 1D Hann window of nth order.
  - Parameters:
    - `m [int]`: Number of points in window (typically the length of a signal).
    - `order [int]`: Filter order.
  - Returns: Window array.
- `hyperspy.misc.math_tools.isfloat(number)`: Check if a number or array is of float type.
  - This is necessary because e.g. `isinstance(np.float32(2), float)` is False.
- `hyperspy.misc.math_tools.order_of_magnitude(number)`: Order of magnitude of the given number.
  - Parameters: `number (float)`.
  - Returns: Order.
  - Return type: Float.
- `hyperspy.misc.math_tools.outer_nd(*vec)`: Calculate outer product of n vectors.
  - Parameters: `vec (vector)`.
  - Returns: Outer product.
  - Return type: `ndarray`.
- `hyperspy.misc.math_tools.symmetrize(a)`: Symmetrize an array.

**hyperspy.misc.model_tools module**

- `class hyperspy.misc.model_tools.current_component_values(component, only_free=False, only_active=False)`: Get current component values.
Convenience class that makes use of __repr__ methods for nice printing in the notebook

```python
class hyperspy.misc.model_tools.current_model_values(model, only_free, only_active, 
component_list=None)
```

**Bases:** object

Convenience class that makes use of __repr__ methods for nice printing in the notebook

**hyperspy.misc.physical_constants module**

**hyperspy.misc.physics_tools module**

```python
hyperspy.misc.physics_tools.bragg_scattering_angle(d, E0=100)
```

Calculate the first order bragg diffraction semiangle.

**Parameters**

- `d` *(float)* – interplanar distance in m.
- `E0` *(float)* – Incident energy in keV

**Returns**

- `float` *(Semiangle of scattering of the first order difracted beam. This is two times the bragg angle.)*

```python
hyperspy.misc.physics_tools.effective_Z(Z_list, exponent=2.94)
```

Effective atomic number of a compound or mixture.

Exponent = 2.94 for X-ray absorption.

**Parameters**

- `Z_list` *(list of tuples)* – A list of tuples (f,Z) where f is the number of atoms of the element in the molecule and Z its atomic number

**Returns**

- `Return type` *float*

**hyperspy.misc.rgb_tools module**

```python
hyperspy.misc.rgb_tools.is_rgb(array)
hyperspy.misc.rgb_tools.is_rgba(array)
hyperspy.misc.rgb_tools.is_rgbx(array)
hyperspy.misc.rgb_tools.regular_array2rgbx(data)
hyperspy.misc.rgb_tools.rgbx2regular_array(data, plot_friendly=False)
```

Transforms a RGBx array into a standard one

**Parameters**

- `data` *(numpy array of RGBx dtype)* –
- `plot_friendly` *(bool)* – If True change the dtype to float when dtype is not uint8 and normalize the array so that it is ready to be plotted by matplotlib.
hyperspy.misc.signal_tools module

hyperspy.misc.signal_tools.are_signals_aligned(*args, ignore_axis=None)

broadcast_signals(*args, ignore_axis=None)

Broadcasts all passed signals according to the HyperSpy broadcasting rules: signal and navigation spaces are each separately broadcasted according to the numpy broadcasting rules. One axis can be ignored and left untouched (or set to be size 1) across all signals.

Parameters

* *args (BaseSignal) – Signals to broadcast together
* ignore_axis (None, str, int, Axis) – The axis to be ignored when broadcasting

Returns

Return type list of signals

hyperspy.misc.slicing module

class hyperspy.misc.slicing.FancySlicing

Create a slice of the signal. The indexing supports integer, decimal numbers or strings (containing a decimal number and an units).

```python
>>> 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])
```

hyperspy.misc.slicing.copy_slice_from_whitelist(_from, _to, dims, both_slices, isNav, order=None)

Copies things from one object to another, according to whitelist, slicing where required.

Parameters

* _from (object) – Original object
* _to (object) – Target object
* dims (tuple) – (navigation_dimensions, signal_dimensions) of the original object that is sliced
• **both_slices** *(tuple)* – (original_slices, array_slices) of the operation that is performed
  • **isNav** *(bool)* – if the slicing operation is performed on navigation dimensions of the object
  • **order** *(tuple, None)* – if given, performs the copying in the order given. If not all attributes given, the rest is random (the order a whitelist.keys() returns them). If given in the object, _slicing_order is looked up.

**hyperspy.misc.test_utils module**

**hyperspy.misc.test_utils.all_warnings()**
Context for use in testing to ensure that all warnings are raised.

```
>>> import warnings
>>> def foo():
...     warnings.warn(RuntimeWarning("bar"))
We raise the warning once, while the warning filter is set to "once".
Hereafter, the warning is invisible, even with custom filters:
>>> with warnings.catch_warnings():
...     warnings.simplefilter('once')
...     foo()
We can now run `foo()` without a warning being raised:
>>> from numpy.testing import assert_warns
>>> foo()
To catch the warning, we call in the help of `all_warnings`:
>>> with all_warnings():
...     assert_warns(RuntimeWarning, foo)
```

**hyperspy.misc.test_utils.assert_deep_almost_equal(actual, expected, *args, **kwargs)**
Assert that two complex structures have almost equal contents. Compares lists, dicts and tuples recursively. Checks numeric values using assert_allclose() and checks all other values with assert_equal(). Accepts additional positional and keyword arguments and pass those intact to assert_allclose() (that’s how you specify comparison precision). :param actual: :type actual: lists, dicts or tuples :param expected: :type expected: lists, dicts or tuples

**hyperspy.misc.test_utils.assert_warns(message=None, category=None)**
Context for use in testing to catch known warnings matching regexes

**Parameters**

• **message** *(list of) strings or compiled regexes* – Regexes for the desired warning to catch

• **category** *(type or list of types)* – Warning categories for the desired warning to catch

**Examples**

```
>>> from skimage import data, img_as_ubyte, img_as_float
>>> with assert_warns(['precision loss']):
...     d = img_as_ubyte(img_as_float(data.coins()))
```

3.1. hyperspy package
Notes

Upon exiting, it checks the recorded warnings for the desired matching pattern(s). Raises a ValueError if any match was not found or an unexpected warning was raised. Allows for three types of behaviors: “and”, “or”, and “optional” matches. This is done to accommodate different build environments or loop conditions that may produce different warnings. The behaviors can be combined. If you pass multiple patterns, you get an orderless “and”, where all of the warnings must be raised. If you use the “|” operator in a pattern, you can catch one of several warnings. Finally, you can use “|AZ” in a pattern to signify it as optional.

```
hyperspy.misc.test_utils.check_running_tests_in_CI()
hyperspy.misc.test_utils.ignore_warning(message=", category=None)
hyperspy.misc.test_utils.sanitize_dict(dictionary)
```

**hyperspy.misc.tv_denoise module**

```
hyperspy.misc.tv_denoise.tv_denoise(im, weight=50, eps=0.0002, keep_type=False, n_iter_max=200)
```

Perform total-variation denoising on 2-d and 3-d images

**Parameters**

- `im` *(ndarray (2d or 3d) of ints, uints or floats)* – input data to be denoised. `im` can be of any numeric type, but it is cast into an ndarray of floats for the computation of the denoised image.

- `weight` *(float, optional)* – denoising weight. The greater weight, the more denoising (at the expense of fidelity to input)

- `eps` *(float, optional)* – relative difference of the value of the cost function that determines the stop criterion. The algorithm stops when:

  \[(E_{(n-1)} - E_n) < \text{eps} \times E_0\]

- `keep_type` *(bool, optional (False))* – whether the output has the same dtype as the input array. keep_type is False by default, and the dtype of the output is np.float

- `n_iter_max` *(int, optional)* – maximal number of iterations used for the optimization.

**Returns**

- `out` – denoised array

**Return type**

- `ndarray`

**Notes**

The principle of total variation denoising is explained in [http://en.wikipedia.org/wiki/Total_variation_denoising](http://en.wikipedia.org/wiki/Total_variation_denoising)

The principle of total variation denoising is to minimize the total variation of the image, which can be roughly described as the integral of the norm of the image gradient. Total variation denoising tends to produce “cartoon-like” images, that is, piecewise-constant images.

This code is an implementation of the algorithm of Rudin, Fatemi and Osher that was proposed by Chambolle in 1.

References

Examples

```python
>>> # 2D example using ascent
>>> import scipy
>>> ascent = scipy.misc.ascent().astype(np.float)
>>> ascent += 0.5 * ascent.std() * np.random.randn(*ascent.shape)
>>> denoised_ascent = tv_denoise(ascent, weight=60)
>>> # 3D example on synthetic data
>>> x, y, z = np.ogrid[0:40, 0:40, 0:40]
>>> mask = (x - 22)**2 + (y - 20)**2 + (z - 17)**2 < 8**2
>>> mask = mask.astype(np.float)
>>> mask += 0.2 * np.random.randn(*mask.shape)
>>> res = tv_denoise_3d(mask, weight=100)
```

hyperspy.misc.utils module

class hyperspy.misc.utils.DictionaryTreeBrowser

```
class hyperspy.misc.utils.DictionaryTreeBrowser (dictionary=None, double_lines=False)

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.

**export**: saves the dictionary in pretty tree printing format in a text file.

**keys**: returns a list of non-private keys.

**as dictionary**: returns a dictionary representation of the object.

**set_item**: easily set items, creating any necessary node on the way.

**add_node**: adds a node.
```

Examples

```python
>>> 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
```
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

```python
>>> 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 ()
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 (valid encoding str)

get_item (item_path, default=None)
Given a path, return it’s value if it exists, or default value if missing.

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)
• **default** – The value to return if the path does not exist.

### Examples

```python
>>> 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')
False
```

**has_item** *(item_path)*

Given a path, return True if it exists.

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)

### Examples

```python
>>> 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')
False
```

**keys** *( )*

Returns a list of non-private keys.

**set_item** *(item_path, value)*

Given the path and value, create the missing nodes in the path and assign to the last one the value

**Parameters**

- **item_path** *(Str)* – A string describing the path with each item separated by a full stops (periods)

### Examples

```python
>>> dict_browser = DictionaryTreeBrowser({})
>>> dict_browser.set_item('First.Second.Third', 3)
>>> dict_browser
First
  Second
    Third = 3
```

**hyperspy.misc.utils.add_scalar_axis** *(signal)*

**hyperspy.misc.utils.attrsetter** *(target, attrs, value)*

Sets attribute of the target to specified value, supports nested attributes. Only creates a new attribute if the object supports such behaviour (e.g. DictionaryTreeBrowser does)
Parameters

- **target** (*object*) –
- **attrs** (*string*) – attributes, separated by periods (e.g. ‘metadata.Signal>Noise_parameters.variance’)
- **value** (*object*) –

Example

First create a signal and model pair:

```python
>>> s = hs.signals.Signal1D(np.arange(10))
>>> m = s.create_model()
>>> m.signal.data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Now set the data of the model with attrsetter >>> attrsetter(m, ‘signal1D.data’, np.arange(10)+2) >>>
self.signal.data.array([2, 3, 4, 5, 6, 7, 8, 9, 10, 10])

The behaviour is identical to >>> self.signal.data = np.arange(10) + 2

**hyperspy.misc.utils.closest_power_of_two** (*n*)

**hyperspy.misc.utils.create_map_objects** (*function, nav_size, iterating_kwargs, **kwargs*)

To be used in _map_iterate of BaseSignal and LazySignal.

Moved to a separate method to reduce code duplication.

**hyperspy.misc.utils.deprecation_warning** (*msg*)

**hyperspy.misc.utils.dummy_context_manager** (*args, **kwargs*)

**hyperspy.misc.utils.ensure_unicode** (*stuff, encoding='utf8', encoding2='latin-1')

**hyperspy.misc.utils.find_subclasses** (*mod, cls*)

Find all the subclasses in a module.

Parameters

- **mod** (*module*) –
- **cls** (*class*) –

Returns

**Return type** dictionary in which key, item = subclass name, subclass

**hyperspy.misc.utils.fsdict** (*nodes, value, dictionary*)

Populates the dictionary ‘dic’ in a file system-like fashion creating a dictionary of dictionaries from the items present in the list ‘nodes’ and assigning the value ‘value’ to the innermost dictionary.

‘dic’ will be of the type: dic[‘node1’][‘node2’][‘node3’]…[‘nodeN’] = value where each node is like a directory that contains other directories (nodes) or files (values)

**hyperspy.misc.utils.generate_axis** (*origin, step, N, index=0*)

Creates an axis given the origin, step and number of channels

Alternatively, the index of the origin channel can be specified.

Parameters

- **origin** (*float*) –
• **step** (*float*) –
• **N** (*number of channels*) –
• **index** (*int*) – index of origin

Returns

**Return type** Numpy array

`hyperspy.misc.utils.get_object_package_info(obj)`

Get info about object package

**Returns** `dic` – Dictionary containing package and package_version (if available)

**Return type** `dict`

`hyperspy.misc.utils.isiterable(obj)`

`hyperspy.misc.utils.iterable_not_string(thing)`

`hyperspy.misc.utils.map_result_construction(signal, inplace, result, ragged, sig_shape=None, lazy=False)`

`hyperspy.misc.utils.multiply(iterable)`

Return product of sequence of numbers.

Equivalent of `functools.reduce(operator.mul, iterable, 1)`.

```python
>>> product([2**8, 2**30])
274877906944
>>> product([])
1
```

`hyperspy.misc.utils.ordinal(value)`

Converts zero or a positive integer (or their string representations) to an ordinal value.

```python
>>> for i in range(1,13):
...     ordinal(i)
...
'1st'
'2nd'
'3rd'
'4th'
'5th'
'6th'
'7th'
'8th'
'9th'
'10th'
'11th'
'12th'
```

```python
>>> for i in (100, '111', '112',1011):
...     ordinal(i)
...
'100th'
'111th'
'112th'
'1011th'
```
Notes

Author: Serdar Tumgoren http://code.activestate.com/recipes/576888-format-a-number-as-an-ordinal/ MIT license

hyperspy.misc.utils.print_html(f_text,f_html)
Print html version when in Jupyter Notebook

hyperspy.misc.utils.rollelem(a,index,to_index=0)
Roll the specified axis backwards, until it lies in a given position.

Parameters

• a (list) – Input list.

• index (int) – The index of the item to roll backwards. The positions of the items do not change relative to one another.

• to_index (int, optional) – The item is rolled until it lies before this position. The default, 0, results in a “complete” roll.

Returns res – Output list.

Return type list

hyperspy.misc.utils.shorten_name(name,req_l)

hyperspy.misc.utils.signal_range_from_roi(signal_range)

hyperspy.misc.utils.slugify(value,valid_variable_name=False)
Normalizes string, converts to lowercase, removes non-alpha characters, and converts spaces to hyphens.
Adapted from Django’s “django/template/defaultfilters.py”.

hyperspy.misc.utils.stack(signal_list,axis=None,new_axis_name='stack_element',lazy=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 instances) –

• axis (None, int, 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.

• new_axis_name (string) – 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, None) – Returns a LazySignal if True. If None, only returns lazy result if at least one is lazy.

Returns signal – signal list

Return type BaseSignal instance (or subclass, determined by the objects in
Examples

```python
>>> 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.misc.utils.

- `stash_active_state(model)`
- `str2num(string, **kargs)`
  Transform a table in string form into a numpy array
  
  **Parameters**

  - `string` (string)

  **Returns**

  - `Return type` numpy array

- `strlist2enumeration(lst)`
- `swapelem(obj, i, j)`
  Swaps element having index `i` with element having index `j` in object `obj` IN PLACE.
  
  E.g. >>> L = ['a', 'b', 'c'] >>> swapelem(L, 1, 2) >>> print(L)
  
  ['a', 'c', 'b']

- `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

```python
>>> 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)
[<BaseSignal, title: , dimensions: (2|2)>,
  <BaseSignal, title: , dimensions: (2|2)>,
  <BaseSignal, title: , dimensions: (2, 2|2)>]
>>> hs.transpose(signal1, signal2, signal3, signal_axes=["Energy"])
```

hyperspy.misc.utils.

- `underline(line, character='\'-\')`
  Return the line underlined.

Module contents

- `hyperspy.models package`
Submodules

hyperspy.models.edsmodel module

class hyperspy.models.edsmodel.EDSModel(spectrum, auto_background=True,
    auto_add_lines=True, *args, **kwargs)

Build and fit a model of an EDS Signal1D.

Parameters

- **spectrum** *(an EDSSpectrum (or any EDSSpectrum subclass) instance.)* -
- **auto_add_lines** *(boolean) – If True, automatically add Gaussians for all X-rays generated in the energy range by an element, using the edsmodel.add_family_lines method.*
- **auto_background** *(boolean) – If True, adds automatically a polynomial order 6 to the model, using the edsmodel.add_polynomial_background method.*
- **extra arguments are passed to the Model creator. (Any)* -

Example

```python
>>> m = s.create_model()
>>> m.fit()
>>> m.fit_background()
>>> m.calibrate_energy_axis('resolution')
>>> m.calibrate_xray_lines('energy', ['Au_Ma'])
>>> m.calibrate_xray_lines('sub_weight','Mn_La', bound=10)
```

add_family_lines *(xray_lines='from_elements')*

Create the Xray-lines instances and configure them appropriately

If a X-ray line is given, all the the lines of the family is added. For instance if Zn Ka is given, Zn Kb is added too. The main lines (alpha) is added to self.xray_lines

Parameters **xray_lines** *(None, 'from_elements', list of string) – If None, if metadata contains xray_lines list of lines use those. If ‘from_elements’, add all lines from the elements contains in metadata. Alternatively, provide an iterable containing a list of valid X-ray lines symbols. (eg. ‘Al_Ka’,‘Zn_Ka’)).*

add_polynomial_background *(order=6)*

Add a polynomial background.

the background is added to self.background_components

Parameters **order** *(int) – The order of the polynomial*

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** *(a complete dictionary of the model, which includes at)*
• least the following fields –

  components [list] a list of dictionaries of components, one per
  _whitelist [dictionary] a dictionary with keys used as references for saved at-
  tributes, for more information, see hypersy.misc.export_dictionary.

  export_to_dictionary() – any field from _whitelist.keys() *

Examples

```python
>>> s = signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> l1 = components1d.Lorentzian()
>>> l2 = components1d.Lorentzian()
>>> m.append(l1)
>>> m.append(l2)
>>> d = m.as_dictionary()
>>> m2 = s.create_model(dictionary=d)
```

calibrate_energy_axis (calibrate='resolution', xray_lines='all_alpha', **kwargs)

Calibrate the resolution, the scale or the offset of the energy axis by fitting.

Parameters

• calibrate ('resolution' or 'scale' or 'offset') – If 'resolution', fits
  the width of Gaussians place at all x-ray lines. The width is given by a model of the
detector resolution, obtained by extrapolating the energy_resolution_MnKa in metadata
metadata. This method will update the value of energy_resolution_MnKa. If 'scale',
calibrate the scale of the energy axis If 'offset', calibrate the offset of the energy axis

• xray_lines (list of str or 'all_alpha') – The Xray lines. If 'all_alpha',
  fit all using all alpha lines

• **kwargs (extra key word arguments) – All extra key word arguments are
  passed to fit or multifit, depending on the value of kind.

calibrate_xray_lines (calibrate='energy', xray_lines='all', bound=1, kind='single', **kwargs)

Calibrate individually the X-ray line parameters.

The X-ray line energy, the weight of the sub-lines and the X-ray line width can be calibrated.

Parameters

• calibrate ('energy' or 'sub_weight' or 'width') – If 'energy', cali-
  brate the X-ray line energy. If 'sub_weight', calibrate the ratio between the main line
alpha and the other sub-lines of the family If 'width', calibrate the X-ray line width.

• xray_lines (list of str or 'all') – The Xray lines. If 'all', fit all lines

• bounds (float) – for 'energy' and 'width' the bound in energy, in eV for 'sub_weight'
  Bound the height of the peak to fraction of its height

• kind ('single', 'multi') – If 'single' fit only the current location. If 'multi'
  use multifit.

• **kwargs (extra key word arguments) – All extra key word arguments are
  passed to fit or multifit, depending on the value of kind.
disable_xray_lines()
Disable the X-ray lines components.

enable_xray_lines()
Enable the X-ray lines components.

fit_background(start_energy=None, end_energy=None, windows_sigma=(4.0, 3.0), kind='single', **kwargs)
Fit the background in the energy range containing no X-ray line.
After the fit, the background is fixed.

Parameters

• start_energy ((float, None)) – If float, limit the range of energies from the left to the given value.
• end_energy ((float, None)) – If float, limit the range of energies from the right to the given value.
• windows_sigma (tuple of two float) – The (lower, upper) bounds around each X-ray line, each as a float, to define the energy range free of X-ray lines.
• kind ({'single', 'multi'}) – If ‘single’ fit only the current location. If ‘multi’ use multifit.
• **kwargs (extra key word arguments) – All extra key word arguments are passed to fit or

See also:
free_background()

fix_background()
Fix the background components.

fix_sub_xray_lines_weight(xray_lines='all')
Fix the weight of a sub X-ray lines to the main X-ray lines
Establish the twin on the height of sub-Xray lines (non alpha)

fix_xray_lines_energy(xray_lines='all')
Fix the X-ray line energy (shift or centre of the Gaussian)

Parameters

• xray_lines (list of str, 'all', or 'all_alpha') – The Xray lines. If ‘all’, fit all lines. If ‘all_alpha’ fit all using all alpha lines.
• bound (float) – the bound around the actual energy, in keV or eV

fix_xray_lines_width(xray_lines='all')
Fix the X-ray line width (sigma of the Gaussian)

Parameters

• xray_lines (list of str, 'all', or 'all_alpha') – The Xray lines. If ‘all’, fit all lines. If ‘all_alpha’ fit all using all alpha lines.
• bound (float) – the bound around the actual energy, in keV or eV

free_background()
Free the yscale of the background components.
**free_sub_xray_lines_weight** \((xray\_lines='all', \text{bound}=0.01)\)
Free the weight of a sub X-ray lines

Remove the twin on the height of sub-Xray lines (non alpha)

**Parameters**

- **xray_lines** (list of str or 'all') – The Xray lines. If 'all', fit all lines
- **bounds** (float) – Bound the height of the peak to a fraction of its height

**free_xray_lines_energy** \((xray\_lines='all', \text{bound}=0.001)\)
Free the X-ray line energy (shift or centre of the Gaussian)

**Parameters**

- **xray_lines** (list of str or 'all') – The Xray lines. If 'all', fit all lines
- **bound** (float) – the bound around the actual energy, in keV or eV

**free_xray_lines_width** \((xray\_lines='all', \text{bound}=0.01)\)
Free the X-ray line width (sigma of the Gaussian)

**Parameters**

- **xray_lines** (list of str or 'all') – The Xray lines. If 'all', fit all lines
- **bound** (float) – the bound around the actual energy, in keV or eV

**get_lines_intensity** \((xray\_lines=None, \text{plot\_result}=False, \text{only\_one}=True, \text{only\_lines}=('a', ), **\text{kwargs})\)
Return the fitted intensity of the X-ray lines.

Return the area under the gaussian corresp to the X-ray lines

**Parameters**

- **xray_lines** ((None, list of string)) – If None, if metadata.Sample.elements.xray_lines contains a list of lines use those. If metadata.Sample.elements.xray_lines is undefined or empty but metadata.Sample.elements is defined, use the same syntax as add_line to select a subset of lines for the operation. Alternatively, provide an iterable containing a list of valid X-ray lines symbols.
- **plot_result** (bool) – If True, plot the calculated line intensities. If the current object is a single spectrum it prints the result instead.
- **only_one** (bool) – If False, use all the lines of each element in the data spectral range. If True use only the line at the highest energy above an overvoltage of 2 (< beam energy / 2).
- **only_lines** ((None, list of strings)) – If not None, use only the given lines.
- **kwargs** – The extra keyword arguments for plotting. See utils.plot.plot_signals

Returns **intensities** – A list containing the intensities as Signal subclasses.

Return type **list**

**Examples**

```python
>>> m.multifit()
>>> m.get_lines_intensity(['C_Ka', 'Ta_Ma'])
```
remove(thing)
Remove component from model.

Examples

```python
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g = hs.model.components1D.Gaussian()
>>> m.append(g)

You could remove g like this

```python
>>> m.remove(g)
```

Like this:

```python
>>> m.remove("Gaussian")
```

Or like this:

```python
>>> m.remove(0)
```

property spectrum
property units_factor

hyperspy.models.edssemmodel module

class hyperspy.models.edssemmodel.EDSSEMModel(spectrum, auto_background=True, auto_add_lines=True, *args, **kwargs)
Bases: hyperspy.models.edsmodel.EDSModel
Build and fit a model to EDS data acquired in the SEM.

Parameters

- spectrum (an EDSSEMSpectrum instance) –
- auto_add_lines (boolean) – If True, automatically add Gaussians for all X-rays generated in the energy range by an element, using the edsmodel.add_family_lines method.
- auto_background (boolean) – If True, adds automatically a polynomial order 6 to the model, using the edsmodel.add_polynomial_background method.
- extra arguments are passed to the Model constructor. (Any) –

hyperspy.models.edstemmodel module

class hyperspy.models.edstemmodel.EDSTEMModel(spectrum, auto_background=True, auto_add_lines=True, *args, **kwargs)
Bases: hyperspy.models.edsmodel.EDSModel
Build and fit a model to EDS data acquired in the TEM.

Parameters

- spectrum (an EDSTEMSpectrum instance) –
• **auto_add_lines** *(boolean)* – If True, automatically add Gaussians for all X-rays generated in the energy range by an element, using the edsmodel.add_family_lines method.

• **auto_background** *(boolean)* – If True, adds automatically a polynomial order 6 to the model, using the edsmodel.add_polynomial_background method.

• **extra arguments are passed to the Model constructor** *(Any)* –

### hyperspy.models.eelsmodel module

#### class hyperspy.models.eelsmodel.EELSModel

```python
class hyperspy.models.eelsmodel.EELSModel(signal1D, auto_background=True, auto_add_edges=True, ll=None, GOS=None, dictionary=None)
```

**Bases:** hyperspy.models.model1d.Model1D

Build an EELS model

**Parameters**

• **spectrum** *(a Signal1D (or any Signal1D subclass) instance)* –

• **auto_background** *(boolean)* – If True, and if spectrum is an EELS instance adds automatically a powerlaw to the model and estimate the parameters by the two-area method.

• **auto_add_edges** *(boolean)* – If True, and if spectrum is an EELS instance, it will automatically add the ionization edges as defined in the Signal1D instance. Adding a new element to the spectrum using the components.EELSSpectrum.add_elements method automatically add the corresponding ionisation edges to the model.

• **ll** *(None, EELSSpectrum)* – If an EELSSpectrum is provided, it will be assumed that it is a low-loss EELS spectrum, and it will be used to simulate the effect of multiple scattering by convolving it with the EELS spectrum.

• **GOS** *(‘hydrogenic’, ‘Hartree-Slater’, None)* – The GOS to use when auto adding core-loss EELS edges. If None it will use the Hartree-Slater GOS if they are available, otherwise it will use the hydrogenic GOS.

• **dictionary** *(dict, None)* – A dictionary to be used to recreate a model. Usually generated using hyperspy.model.as_dictionary()

#### append(component)

Add component to Model.

**Parameters**

**thing** *(Component instance.)* –

#### disable_background()

Disable the background components.

#### disable_edges(edges_list=None)

Disable the edges listed in edges_list. If edges_list is None (default) all the edges with onset in the spectrum energy region will be disabled.

**Parameters**

**edges_list** *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

**See also:**

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(),
enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

disable_fine_structure(edges_list=None)
 Disable the fine structure of the edges listed in edges_list. If edges_list is None (default) the fine structure of all the edges with onset in the spectrum energy region will be disabled.

Parameters edges_list (None or list of EELSCLEdge or list of edge names) – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

disable_free_onset_energy(edges_list=None)
 Disable the automatic freeing of the onset_energy parameter during a smart fit for the edges listed in edges_list. If edges_list is None (default) the onset_energy of all the edges with onset in the spectrum energy region will not be freed. Note that if their attribute edge.onset_energy.free is True, the parameter will be free during the smart fit.

Parameters edges_list (None or list of EELSCLEdge or list of edge names) – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

enable_background()
 Enable the background components.

enable_edges(edges_list=None)
 Enable the edges listed in edges_list. If edges_list is None (default) all the edges with onset in the spectrum energy region will be enabled.

Parameters edges_list (None or list of EELSCLEdge or list of edge names) – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

enable_fine_structure(edges_list=None)
 Enable the fine structure of the edges listed in edges_list. If edges_list is None (default) the fine structure of all the edges with onset in the spectrum energy region will be enabled.
Parameters `edges_list` *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

```
enable_edges(), disable_edges(), enable_background(),
disable_background(), enable_fine_structure(), disable_fine_structure(),
set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(),
enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(),
free_edges(), fix_fine_structure(), free_fine_structure()
```

**enable_free_onset_energy** *(edges_list=None)*

Enable the automatic freeing of the `onset_energy` parameter during a smart fit for the edges listed in `edges_list`. If `edges_list` is None (default) the `onset_energy` of all the edges with onset in the spectrum energy region will be freed.

Parameters `edges_list` *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

```
enable_edges(), disable_edges(), enable_background(),
disable_background(), enable_fine_structure(), disable_fine_structure(),
set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(),
enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(),
free_edges(), fix_fine_structure(), free_fine_structure()
```

**fit** *(fitter=None, method='ls', grad=False, bounded=False, ext_bounding=False, update_plot=False, kind='std', **kwargs)*

Fits the model to the experimental data

Parameters

- **fitter** *(None, "leastsq", "odr", "mpfit", "fmin")* – The optimizer to perform the fitting. If None the fitter defined in the Preferences is used. leastsq is the most stable but it does not support bounding. mpfit supports bounding. fmin is the only one that supports maximum likelihood estimation, but it is less robust than the Levenberg–Marquardt based leastsq and mpfit, and it is better to use it after one of them to refine the estimation.

- **method** *(\{'ls', \'ml'\})* – Choose ‘ls’ (default) for least squares and ‘ml’ for maximum-likelihood estimation. The latter only works with fitter = ‘fmin’.

- **grad** *(bool)* – If True, the analytical gradient is used if defined to speed up the estimation.

- **ext_bounding** *(bool)* – If True, enforce bounding by keeping the value of the parameters constant out of the defined bounding area.

- **bounded** *(bool)* – If True performs bounded optimization if the fitter supports it. Currently only mpfit support bounding.

- **update_plot** *(bool)* – If True, the plot is updated during the optimization process. It slows down the optimization but it permits to visualize the optimization evolution.

- **kind** *(\{'std', \'smart'\})* – If ‘std’ (default) performs standard fit. If ‘smart’ performs smart_fit

- ****kwargs** *(key word arguments)* – Any extra key word argument will be passed to the chosen fitter
See also:

```
multifit(), smart_fit()
```

**fit_background** *(start_energy=None, only_current=True, **kwargs)*

Fit the background to the first active ionization edge in the energy range.

**Parameters**

- **start_energy** *(float, None, optional)* – If float, limit the range of energies from the left to the given value. Default None.

- **only_current** *(bool, optional)* – If True, only fit the background at the current coordinates. Default True.

- ****kwargs **(extra key word arguments)** – All extra key word arguments are passed to fit or multifit.

**fix_edges** *(edges_list=None)*

Fixes all the parameters of the edges given in edges_list. If edges_list is None (default) all the edges will be fixed.

**Parameters edges_list** *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

```
enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()
```

**fix_fine_structure** *(edges_list=None)*

Fixes all the parameters of the edges given in edges_list. If edges_list is None (default) all the edges will be fixed.

**Parameters edges_list** *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

```
enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()
```

**free_edges** *(edges_list=None)*

Frees all the parameters of the edges given in edges_list. If edges_list is None (default) all the edges will be freed.

**Parameters edges_list** *(None or list of EELSCLEdge or list of edge names)* – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

```
enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(),
```

---

414 Chapter 3. hyperspy
enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

free_fine_structure(edges_list=None)
Frees all the parameters of the edges given in edges_list. If edges_list is None (default) all the edges will be freed.

Parameters:
edges_list (None or list of EELSCLEdge or list of edge names) – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()

quantify()
Prints the value of the intensity of all the independent active EELS core loss edges defined in the model

remove(component)
Remove component from model.

Examples

```python
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g = hs.model.components1D.Gaussian()
>>> m.append(g)
```

You could remove `g` like this

```python
>>> m.remove(g)
```

Like this:

```python
>>> m.remove("Gaussian")
```

Or like this:

```python
>>> m.remove(0)
```

remove_fine_structure_data(edges_list=None)
Remove the fine structure data from the fitting routine as defined in the fine_structure_width parameter of the component.EELSCLEdge

Parameters:
edges_list (None or list of EELSCLEdge or list of edge names) – If None, the operation is performed on all the edges in the model. Otherwise, it will be performed only on the listed components.

See also:

enable_edges(), disable_edges(), enable_background(), disable_background(), enable_fine_structure(), disable_fine_structure(), set_all_edges_intensities_positive(), unset_all_edges_intensities_positive(), enable_free_onset_energy(), disable_free_onset_energy(), fix_edges(), free_edges(), fix_fine_structure(), free_fine_structure()
**resolve_fine_structure** *(preedge_safe_window_width=2, i1=0)*

Adjust the fine structure of all edges to avoid overlapping.

This function is called automatically everytime the position of an edge changes.

**Parameters**

- **preedge_safe_window_width** *(float)* – minimum distance between the fine structure of an ionization edge and that of the following one. Default 2 (eV).

**resume_auto_fine_structure_width** *(update=True)*

Enable the automatic adjustment of the core-loss edges fine structure width.

**Parameters**

- **update** *(bool, optional)* – If True, also execute the automatic adjustment (default).

**See also:**

suspend_auto_fine_structure_width()

**set_all_edges_intensities_positive** *

**property signal1D**

**smart_fit** *(start_energy=None, **kwargs)*

Fits everything in a cascade style.

**Parameters**

- **start_energy** *(float, None)* – If float, limit the range of energies from the left to the given value.
- ****kwargs** *(key word arguments)* – Any extra key word argument will be passed to the fit method. See the fit method documentation for a list of valid arguments.

**See also:**

fit(), multifit()

**suspend_auto_fine_structure_width** *

Disable the automatic adjustament of the core-loss edges fine structure width.

**See also:**

resume_auto_fine_structure_width()

**two_area_background_estimation** *(E1=None, E2=None, powerlaw=None)*

Estimates the parameters of a power law background with the two area method.

**Parameters**

- **E1** *(float)* –
- **E2** *(float)* –
- **powerlaw** *(PowerLaw component or None)* – If None, it will try to guess the right component from the background components of the model.

**unset_all_edges_intensities_positive** *

**hyperspy.models.model1d module**

**class hyperspy.models.model1d.ComponentFit** *(model, component, signal_range=None, estimate_parameters=True, fit_independent=False, only_current=True, **kwargs)*

Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
apply()

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 strings 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.models.model1d.Model1D(signal1D, dictionary=None)
    Bases: hyperspy.model.BaseModel

    Model and data fitting for one dimensional signals.

    A model is constructed as a linear combination of components that are added to the model using
    append() or extend(). There are many predefined components available in the
    components module. 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.

    signal
    It contains the data to fit.
    
    Type  Signal1D instance

    chisq
    Chi-squared of the signal (or np.nan if not yet fit)
    
    Type  A Signal of floats

    dof
    Degrees of freedom of the signal (0 if not yet fit)
    
    Type  A Signal of integers

    red_chisq
    Reduced chi-squared.
    
    Type  Signal instance

    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.
    
    Type  ModelComponents instance

append()
    Append one component to the model.

extend()
    Append multiple components to the model.

remove()
    Remove component from model.
as_signal()  
Generate a Signal1D instance (possible multidimensional) from the model.

store_current_values()  
Store the value of the parameters at the current position.

fetch_stored_values()  
fetch stored values of the parameters.

update_plot()  
Force a plot update. (In most cases the plot should update automatically.)

set_signal_range, remove_signal_range, reset_signal_range, add_signal_range.  
Customize the signal range to fit.

fit, multifit  
Fit the model to the data at the current position or the full dataset.

save_parameters2file, load_parameters_from_file  
Save/load the parameter values to/from a file.

plot()  
Plot the model and the data.

enable_plot_components, disable_plot_components  
Plot each component separately. (Use after plot.)

set_current_values_to()  
Set the current value of all the parameters of the given component as the value for all the dataset.

export_results()  
Save the value of the parameters in separate files.

plot_results()  
Plot the value of all parameters at all positions.

print_current_values()  
Print the value of the parameters at the current position.

enable_adjust_position, disable_adjust_position  
Enable/disable interactive adjustment of the position of the components that have a well defined position.  
(Use after plot).

fit_component()  
Fit just the given component in the given signal range, that can be set interactively.

set_parameters_not_free, set_parameters_free  
Fit the free status of several components and parameters at once.

set_parameters_value()  
Set the value of a parameter in components in a model to a specified value.

as_dictionary()  
Exports the model to a dictionary that can be saved in a file.

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.
```python
>>> 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()
Components Parameter Value
Gaussian
    sigma 1.000000
    A 1.000000
    centre 0.000000
>>> g.centre.value = 3
>>> m.print_current_values()
Components Parameter Value
Gaussian
    sigma 1.000000
    A 1.000000
    centre 3.000000
>>> g.sigma.value
1.0
>>> m.fit()
>>> g.sigma.value
1.9779042300856682
>>> m[0].sigma.value
1.9779042300856682
>>> m["Gaussian"].centre.value
-0.072121936813224569
```

### 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** *(None or float)* –
- **x2** *(None or float)* –

### append(thing)

Add component to Model.

#### Parameters

- **thing** *(Component instance)* –

### disable_adjust_position()

Disables the interactive adjust position feature

#### See also:

- **enable_adjust_position()**

### disable_plot_components()

### 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 components))* – 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.

3.1. hyperspy package

419
• **fixThem** *(bool)* – 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.

• **showLabel** *(bool, optional)* – 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()

* fit_component*(component, signal_range='interactive', estimate_parameters=True, fit_independent=False, only_current=True, display=True, toolkit=None, **kwargs)*

Fit just 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 instance)* – The component must be in the model, otherwise an exception is raised. The component can be specified by name, index or itself.

• **signal_range** *(\{'interactive\', (left_value, right_value), 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. If None the current signal range is used.

• **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 strings 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.

**Examples**

Signal range set interactively

```python
>>> 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

```python
>>> m.fit_component(g1, signal_range=(1,7))
```
property low_loss

plot(plot_components=False, **kwargs)
Plots 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.
- **kwargs** – All extra keyword arguments are passed to Signal1D.plot

remove(things)
Remove component from model.

Examples

```python
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g = hs.model.components1D.Gaussian()
>>> m.append(g)
```

You could remove g like this

```python
>>> m.remove(g)
```

Like this:

```python
>>> m.remove("Gaussian")
```

Or like this:

```python
>>> 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**(None or float) –
- **x2**(None or float) –

reset_signal_range()
Resets the data range

reset_the_signal_range()

set_convolution_axis()
Creates an axis to use to generate the data of the model in the precise scale to obtain the correct axis and origin after convolution with the lowloss spectrum.

set_signal_range(*args, **kwargs)
Use only the selected spectral range defined in its own units in the fitting routine.

Parameters

- **E1**(None or float) –
- **E2**(None or float) –

property signal
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 many predefined components available in the components2D module. 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.

Note that methods are not yet defined for plotting 2D models or using gradient based optimisation methods - these will be added soon.

- **signal**
  - It contains the data to fit.
    - Type: Signal2D instance

- **chisq**
  - Chi-squared of the signal (or np.nan if not yet fit)
    - Type: A Signal of floats

- **dof**
  - Degrees of freedom of the signal (0 if not yet fit)
    - Type: A Signal of integers

- **red_chisq**
  - Reduced chi-squared.
    - Type: Signal instance

- **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.
    - Type: `ModelComponents` instance

- **append()**
  - Append one component to the model.

- **extend()**
  - Append multiple components to the model.

- **remove()**
  - Remove component from model.

- **fit, multifit**
  - Fit the model to the data at the current position or the full dataset.

See also:

Base, Model1D
Example

```python
add_signal_range(*args, **kwargs)
disable_adjust_position()
enable_adjust_position(components=None, fix_them=True, show_label=True)
plot(plot_components=False)
remove_signal_range(*args, **kwargs)
reset_signal_range()
reset_the_signal_range()
set_signal_range(*args, **kwargs)

property signal
```

Module contents

**hyperspy.samfire_utils package**

Subpackages

**hyperspy.samfire_utils.goodness_of_fit_tests package**

Submodules

**hyperspy.samfire_utils.goodness_of_fit_tests.information_theory module**

```python
class hyperspy.samfire_utils.goodness_of_fit_tests.information_theory.AIC_test(tolerance)
    Bases: object
    map(model, mask)
    test(model, ind)

class hyperspy.samfire_utils.goodness_of_fit_tests.information_theory.AICc_test(tolerance)
    Bases: object
    map(model, mask)
    test(model, ind)

class hyperspy.samfire_utils.goodness_of_fit_tests.information_theory.BIC_test(tolerance)
    Bases: object
    map(model, mask)
    test(model, ind)
```

**hyperspy.samfire_utils.goodness_of_fit_tests.information_theory.notexp_o(x)**
hyperspy.samfire_utils.goodness_of_fit_tests.red_chisq module

class hyperspy.samfire_utils.goodness_of_fit_tests.red_chisq.red_chisq_test(toleration)
Bases: hyperspy.samfire_utils.goodness_of_fit_tests.test_general.goodness_test

def map(model, mask)
def test(model, ind)

hyperspy.samfire_utils.goodness_of_fit_tests.test_general module

class hyperspy.samfire_utils.goodness_of_fit_tests.test_general.goodness_test
Bases: object

property tolerance

Module contents

hyperspy.samfire_utils.segmenters package

Submodules

hyperspy.samfire_utils.segmenters.histogram module

class hyperspy.samfire_utils.segmenters.histogram.HistogramSegmenter(bins='freedman')
Bases: object

Historam Segmenter strategy of the SAMFire. Uses histograms to estimate parameter distributions, and then passes the most frequent values as the starting parameter estimates.

most_frequent()
Calculates the most frequent values in the currently stored histograms of the database. Does to by looking for local maxima in the frequencies.

update(value_dict)
Recalculates the database, given value dictionary (with all values!)

Parameters value_dict (dict) – dictionary of all already calculated values in the form of {component_name: {parameter_name: values, ...}, ...}

Module contents

hyperspy.samfire_utils.weights package

Submodules

hyperspy.samfire_utils.weights.red_chisq module

class hyperspy.samfire_utils.weights.red_chisq.ReducedChiSquaredWeight
Bases: object

function(ind)
map(mask, slices=slice(None, None, None))

Module contents

Submodules

hyperspy.samfire_utils.fit_tests module

hyperspy.samfire_utils.global_strategies module
class hyperspy.samfire_utils.global_strategies.HistogramStrategy(bins='freedman')
   Bases: hyperspy.samfire_utils.strategy.GlobalStrategy

hyperspy.samfire_utils.local_strategies module
class hyperspy.samfire_utils.local_strategies.ReducedChiSquaredStrategy
   Bases: hyperspy.samfire_utils.strategy.LocalStrategy
   Reduced chi-squared Local strategy of the SAMFire. Uses reduced chi-squared as the weight, and exponential
decay as the decay function.

hyperspy.samfire_utils.samfire_kernel module

ehyperspy.samfire_utils.samfire_kernel.multi_kernel(ind, m_dic, values,
   optional_components, _args, result_q, test_dict)

ehyperspy.samfire_utils.samfire_kernel.single_kernel(model, ind, values,
   optional_components, _args, test)

hyperspy.samfire_utils.samfire_pool module
class hyperspy.samfire_utils.samfire_pool.SamfirePool(**kwargs)
   Bases: hyperspy.utils.parallel_pool.ParallelPool
   Creates and manages a pool of SAMFire workers. For based on ParallelPool - either creates processes using
   multiprocesing, or connects and sets up ippyparallel 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.
   prepare_workers()
   given SAMFire object, populates the workers with the required information. In case of multiprocessing,
   starts worker listening to the queues.
update_parameters()
    updates various worker parameters

ping_workers()
    pings all workers. Stores the one-way trip time and the process_id (pid) of each worker if available

add_jobs()
    adds the requested number of jobs to the queue

parse()
    parses the messages, returned from the workers.

collect_results()
    collects all currently available results and parses them

run()
    runs the full procedure until no more pixels are left to run in the SAMFire

stop()
    stops the pool, (for ipyparallel) clears the memory

setup()
    sets up the ipyparallel or multiprocessing pool (collects to the client or creates the pool)

sleep()
    sleeps for the specified time, by default timestep

has_pool
    Boolean if the pool is available and active
        Type Bool

pool
    The pool object
        Type {ipyparallel.load_balanced_view, multiprocessing.Pool}

ipython_kwargs
    The dictionary with Ipyparallel connection arguments.
        Type dict

timeout
    Timeout for either pool when waiting for results
        Type float

num_workers
    The number of workers actually created (may be less than requested, but can’t be more)
        Type int

timestep
    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.
        Type float

ping
    If recorded, stores one-way trip time of each worker
        Type dict

pid
    If available, stores the process-id of each worker
Type `dict`

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` *(int)*
  - The number of jobs to add. If None (default), adds `need_pixels` to the queue.

**collect_results** *(timeout=None)*

Collects and parses all results, currently not processed due to being in the queue.

**Parameters**
- `timeout` *(float)*
  - the time to wait when collecting results. If None, the default timeout is used.

**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)*

Parses the value, returned from the workers.

**Parameters**
- `value` *(tuple)*
  - `value` 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:
  - `(worker_id, pid, pong_time, optional_message_str)`
  - `(worker_id, error_message_string)`
  - `(worker_id, pixel_index, result_dict, bool_if_result_converged)`

**ping_workers** *(timeout=None)*

Pings the workers and records one-way trip time and (if available) pid of the worker.

**Parameters**
- `timeout` *(float)*
  - the time to wait when collecting results after sending out the ping. If None, the default timeout is used.

**prepare_workers** *(samfire)*

Prepares the workers for work, in case of multiprocessing starts listening.

**Parameters**
- `samfire` *(samfire)*
  - the SAMFire object that will be using the pool

**run**

Runs 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.

**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
hyperspy.samfire_utils.samfire_worker module

class hyperspy.samfire_utils.samfire_worker.Worker(identity, individual_queue=None, shared_queue=None, result_queue=None)

Bases: object

change_timestep (value)

compare_models()

create_model (signal_dict, model_letter)

fit (component_comb)

generate_component_combinations()

generate_values_iterator (turned_on_names)

listen()

parse (result)

ping (message=None)

reset()

run_pixel (ind, value_dict)

send_results (current=False)

set_optional_names (optional_names)

set_parameter_boundaries (received)

set_values (name_list, iterator)

setup_test (test_string)

sleep (howlong=None)

start_listening()

stop_listening()

hyperspy.samfire_utils.samfire_worker.create_worker (identity, individual_queue=None, shared_queue=None, result_queue=None)

hyperspy.samfire_utils.strategy module

class hyperspy.samfire_utils.strategy.GlobalStrategy (name)

Bases: hyperspy.samfire_utils.strategy.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, HistogramTilePlot)) – If given updates the plot.
refresh(overwrite, given_pixels=None)
    Refreshes the database (i.e. constructs it again from scratch)

segmenter = None

values(ind=None)
    Returns the saved most frequent values that should be used for prediction

class hyperspy.samfire_utils.strategy.LocalStrategy(name)
    Bases: hyperspy.samfire_utils.strategy.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 (Image, None) – if an already plotted image, then updates. Otherwise creates a new one.

    Returns
    fig – the resulting image. If passed again, will be updated (computationally cheaper operation).

    Return type
    Image

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 (boolean numpy array) – Pixels with True value are assumed as correctly calculated.

property samf
    The SAMFire that owns this strategy.

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 – A dictionary of estimates, structured as {component_name: {parameter_name: value, ...}, ...} for active components and free parameters.

    Return type
    dict

property weight
    A Weight object, able to assign significance weights to separate pixels or maps, given the model.

class hyperspy.samfire_utils.strategy.SamfireStrategy
    Bases: object

A SAMFire strategy base class.
close_plot = None
name = ''
remove()
  Removes this strategy from its SAMFire
samf = None
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.

hyperspy.samfire_utils.strategy.make_sure_ind(inds, req_len=None)
  Given an object, constructs a tuple of floats the required length. Either removes items that cannot be cast as
  floats, or adds the last valid item until the required length is reached.

Parameters
  • inds (sequence) – the sequence to be constructed into tuple of floats
  • req_len ({None, number}) – The required length of the output

Returns
  indices

Return type
  tuple of floats

hyperspy.samfire_utils.strategy.nearest_indices(shape, ind, radii)
  Returns the slices to slice a given size array to get the required size rectangle around the given index. Deals
  nicely with boundaries.

Parameters
  • shape (tuple) – the shape of the original (large) array
  • ind (tuple) – the index of interest in the large array (centre)
  • radii (tuple of floats) – the distances of interests in all dimensions around the
    centre index.

Returns
  • slices (tuple of slices) – The slices to slice the large array to get the required region.
  • center (tuple of ints) – The index of the original centre (ind) position in the new (sliced)
    array.

Module contents

hyperspy.tests package

Subpackages

hyperspy.tests.axes package

Submodules
hyperspy.tests.axes.test_axes_manager module

class hyperspy.tests.axes.test_axes_manager.TestAxesHotkeys
    Bases: object
    setup_method(method)
    test_hotkeys_in_six_dimensions()
        Step twice increasing and once decreasing all axes

class hyperspy.tests.axes.test_axes_manager.TestAxesManager
    Bases: object
    setup_method(method)
    test_reprs()
    test_update_from()

class hyperspy.tests.axes.test_axes_manager.TestAxesManagerExtent
    Bases: object
    test_1d_basesignal()
    test_1d_signal1d()
    test_2d_signal1d()
    test_2d_signal2d()
    test_3d_signal1d()
    test_3d_signal2d()
    test_changing_scale_offset()

class hyperspy.tests.axes.test_axes_manager.TestAxesManagerScaleOffset
    Bases: object
    test_change_offset()
    test_change_offset_scale()
    test_change_scale()
    test_low_high_value()

hyperspy.tests.axes.test_axes_manager.test_setting_indices_coordinates()

hyperspy.tests.axes.test_conversion_units module

hyperspy.tests.axes.test_data_axis module

Module contents

hyperspy.tests.component package

Submodules
hyperspy.tests.component.test_bleasdale module

class hyperspy.tests.component.test_bleasdale.TestFunction

hyperspy.tests.component.test_component module

class hyperspy.tests.component.test_component.TestCallMethods
    Bases: object
    setup_method(method)
    test_call()
    test_plotting_active_component_binned()
    test_plotting_active_component_notbinned()
    test_plotting_active_component_out_of_range()
    test_plotting_not_active_component()

class hyperspy.tests.component.test_component.TestGeneralMethods
    Bases: object
    setup_method(method)
    test_export_all_no_twins()
    test_export_all_twins()
    test_export_free()
    test_fetch_from_array()
    test_fetch_from_array_free()
    test_fetch_stored_values_all()
    test_fetch_stored_values_all_twinned()
    test_fetch_stored_values_all_twinned_bad()
    test_fetch_stored_values_fixed()
    test_set_parameters_free_all()
    test_set_parameters_free_name()
    test_set_parameters_not_free_all()
    test_set_parameters_not_free_name()
    test_update_number_parameters()

class hyperspy.tests.component.test_component.TestMultidimensionalActive
    Bases: object
    setup_method(method)
    test_disable_pixel_switching()
    test_disable_pixel_switching_current_off()
    test_disable_pixel_switching_current_on()
    test_enable_pixel_switching_current_off()
test_enable_pixel_switching_current_on()

def test_update_number_free_parameters():
    pass

class TestParametersAsSignals(hyperspy.tests.component.test_component_active_array.TestParametersAsSignals):
    def setup_method(self, method):
        pass

def test_always_active():
    pass
def test_some_inactive():
    pass
def test_stash_array():
    pass

class TestSetParameters(hyperspy.tests.component.test_component_set_parameters.TestSetParameters):
    def setup_method(self, method):
        pass
def test_set_parameters_free1():
    pass
def test_set_parameters_free2():
    pass
def test_set_parameters_not_free1():
    pass
def test_set_parameters_not_free2():
    pass

class TestExpression2D(hyperspy.tests.component.test_components2D.TestExpression2D):
    def setup_method(self, method):
        pass
def test_no_function_nd():
    pass
def test_no_function_nd_signal():
    pass
def test_no_rotation():
    pass
def test_with_rotation():
    pass
def test_with_rotation_center_tuple():
    pass
def test_with_rotation_no_position():
    pass
def test_with_rotation_no_position_init_values():
    pass
class TestGaussian2D(hyperspy.tests.component.test_components2D.TestGaussian2D):
    def setup_method(self, method):
        pass
HyperSpy Documentation, Release 1.5.1.dev

    setup_method(method)
    test_values()

hyperspy.tests.component.test_double_power_law module

    hyperspy.tests.component.test_double_power_law.test_function()

hyperspy.tests.component.test_erf module

hyperspy.tests.component.test_exponential module

    hyperspy.tests.component.test_exponential.test_function()

hyperspy.tests.component.test_gaussian module

hyperspy.tests.component.test_gaussian2d module

    hyperspy.tests.component.test_gaussian2d.test_function()
    hyperspy.tests.component.test_gaussian2d.test_util_fwhm_get()
    hyperspy.tests.component.test_gaussian2d.test_util_fwhm_getset()
    hyperspy.tests.component.test_gaussian2d.test_util_fwhm_set()

hyperspy.tests.component.test_gaussianhf module

hyperspy.tests.component.test_logistic module

    hyperspy.tests.component.test_logistic.test_function()

hyperspy.tests.component.test_lorentzian module

    hyperspy.tests.component.test_lorentzian.test_function()
    hyperspy.tests.component.test_lorentzian.test_util_fwhm_get()
    hyperspy.tests.component.test_lorentzian.test_util_fwhm_getset()
    hyperspy.tests.component.test_lorentzian.test_util_fwhm_set()
    hyperspy.tests.component.test_lorentzian.test_util_gamma_getset()

hyperspy.tests.component.test_powerlaw module

hyperspy.tests.component.test_rc module

    hyperspy.tests.component.test_rc.test_function()
hyperspy.tests.component.test_skewnormal module

hyperspy.tests.component.test_volume_plasmon_drude module

hyperspy.tests.component.test_volume_plasmon_drude.test_function()

Module contents

hyperspy.tests.datasets package

Submodules

hyperspy.tests.datasets.test_artificial_data module

hyperspy.tests.datasets.test_artificial_data.test_get_atomic_resolution_tem_signal2d()
hyperspy.tests.datasets.test_artificial_data.test_get_core_loss_eels_line_scan_signal()
hyperspy.tests.datasets.test_artificial_data.test_get_core_loss_eels_model()
hyperspy.tests.datasets.test_artificial_data.test_get_core_loss_eels_signal()
hyperspy.tests.datasets.test_artificial_data.test_get_low_loss_eels_line_scan_signal()
hyperspy.tests.datasets.test_artificial_data.test_get_low_loss_eels_signal()

hyperspy.tests.datasets.test_eelsdb module

Module contents

hyperspy.tests.drawing package

Submodules

hyperspy.tests.drawing.test_figure module
hyperspy.tests.drawing.test_mpl_testing_setup module
hyperspy.tests.drawing.test_plot_markers module
hyperspy.tests.drawing.test_plot_model module
hyperspy.tests.drawing.test_plot_model1d module
hyperspy.tests.drawing.test_plot_mva module
hyperspy.tests.drawing.test_plot_roi_widgets module
hyperspy.tests.drawing.test_plot_signal module

hyperspy.tests.drawing.test_plot_signal1d module

hyperspy.tests.drawing.test_plot_signal2d module

hyperspy.tests.drawing.test_plot_signal_tools module

hyperspy.tests.drawing.test_plot_widgets module

hyperspy.tests.drawing.test_utils module

Module contents

hyperspy.tests.io package

Submodules

hyperspy.tests.io.generate_dm_testing_files module

Creates Digital Micrograph scripts to generate the dm3 testing files

hyperspy.tests.io.generate_dm_testing_files.generate_1D_files(f, data_types, dmversion)

hyperspy.tests.io.generate_dm_testing_files.generate_2D_files(f, data_types, dmversion)

hyperspy.tests.io.generate_dm_testing_files.generate_3D_files(f, data_types, dmversion)

hyperspy.tests.io.generate_dm_testing_files.generate_4D_files(f, data_types, dmversion)

hyperspy.tests.io.test_blockfile module

hyperspy.tests.io.test_bruker module

hyperspy.tests.io.test dens module

hyperspy.tests.io.test_dm3 module

hyperspy.tests.io.test_dm_stackbuilder_plugin module

class hyperspy.tests.io.test_dm_stackbuilder_plugin.TestStackBuilder
    Bases: object

    test_load_stackbuilder_imagestack()
class hyperspy.tests.io.test_msa.TestExample1
    Bases: object
    setup_method(method)
    test_data()
    test_metadata()
    test_parameters()
    test_write_load_cycle()

class hyperspy.tests.io.test_msa.TestExample1WrongDate
    Bases: object
    setup_method(method)
    test_metadata()

class hyperspy.tests.io.test_msa.TestExample2
    Bases: object
    setup_method(method)
    test_data()
    test_metadata()
    test_parameters()
    test_write_load_cycle()

hyperspy.tests.io.test_msa.test_minimum_metadata_example()
 hyperspy.tests.io.test_tiff module

Module contents

 hyperspy.tests.misc package

 Submodules

 hyperspy.tests.misc.test_arraytools module

 hyperspy.tests.misc.test_date_time_tools module

 hyperspy.tests.misc.test_date_time_tools. test_ISO_format_to_serial_date()
 hyperspy.tests.misc.test_date_time_tools. test_datetime_to_serial_date()
 hyperspy.tests.misc.test_date_time_tools. test_get_date_time_from_metadata()
 hyperspy.tests.misc.test_date_time_tools. test_serial_date_to_ISO_format()
 hyperspy.tests.misc.test_date_time_tools. test_update_date_time_in_metadata()

 hyperspy.tests.misc.test_fei_stream_readers module

 hyperspy.tests.misc.test_image_tools module

 hyperspy.tests.misc.test_math_tools module

 hyperspy.tests.misc.test_math_tools. test_isfloat_float()
 hyperspy.tests.misc.test_math_tools. test_isfloat_int()
 hyperspy.tests.misc.test_math_tools. test_isfloat_npfloating()
 hyperspy.tests.misc.test_math_tools. test_isfloat_npint()

 hyperspy.tests.misc.test_rgbtools module

 class hyperspy.tests.misc.test_rgbtools. TestRGBTools
      Bases: object

 setup_method(method)

 test_regular_array2rgbx_order_from_c()
 test_regular_array2rgbx_order_from_c_slices()
 test_regular_array2rgbx_order_from_f()
 test_regular_array2rgbx_ordermask_from_cmasked()
 test_regular_array2rgbx_ordermask_from_cmasked_slices()
 test_rgbx2regular_array_order_from_c()
 test_rgbx2regular_array_order_from_c_slices()
test_rgbx2regular_array_corder_from_f()

hyperspy.tests.misc.test_test_utils module

docs: hyperspy.tests.misc.test_test_utils
tests:

test_assert_warns_full_message()
test_assert_warns_message_fails()
test_assert_warns_partial_message()
test_assert_warns_regex_message()
test_assert_warns_type()
test_assert_warns_type_fails()
test_ignore_full_message()
test_ignore_message_fails()
test_ignore_partial_message()
test_ignore_regex_message()
test_ignore_type()
test_ignore_type_fails()

warnsA()
warnsB()
warnsC()

hyperspy.tests.misc.test_utils module

docs: hyperspy.tests.misc.test_utils
tests:

test_signal_range_from_roi()
test_slugify()

Module contents

hyperspy.tests.model package

Submodules

hyperspy.tests.model.test_chi_squared module

class hyperspy.tests.model.test_chi_squared.TestChiSquared
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_chisq()
test_chisq_in_range()
test_chisq_with_fit()
test_chisq_with_inactive_components()
test_dof_with_fit()
test_dof_with_inactive_components()
test_dof_with_p0()
test_lazy_chisq()
test_lazy_chisq_in_range()
test_lazy_chisq_with_fit()
test_lazy_chisq_with_inactive_components()
test_lazy_dof_with_fit()
test_lazy_dof_with_inactive_components()
test_lazy_dof_with_p0()
test_lazy_red_chisq()
test_lazy_red_chisq_with_fit()
test_red_chisq()
test_red_chisq_with_fit()

hyperspy.tests.model.test_edsmodel module

hyperspy.tests.model.test_eelsmodel module

hyperspy.tests.model.test_fancy_indexing module

class hyperspy.tests.model.test_fancy_indexing.TestEELSModelSlicing
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_slicing_low_loss_inav()
    test_lazy_slicing_low_loss_isig()
    test_slicing_low_loss_inav()
    test_slicing_low_loss_isig()

class hyperspy.tests.model.test_fancy_indexing.TestModelIndexing
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_component_copying_order()
    test_lazy_component_copying_order()
test_lazy_model_navigation_indexer_slice()
test_lazy_model_signal_indexer_slice()
test_model_navigation_indexer_slice()
test_model_signal_indexer_slice()

class hyperspy.tests.model.test_fancy_indexing.TestModelIndexingClass
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_model_class()
    test_model_class()

test_lazy_model_class()
test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()

test_model_class()
```
test_set_parameter_in_model_not_free()
```

test_set_parameter_value module

```python
class hyperspy.tests.model.test_set_parameter_value.TestSetParameterInModel
    Bases: object
    setup_method(method)
    test_set_active_value1()
    test_set_active_value2()
    test_set_active_value3()
    test_set_parameter_value1()
    test_set_parameter_value2()
    test_set_parameter_value3()
```

Module contents

hyperspy.tests.mva package

Submodules

hyperspy.tests.mva.test_bss module

hyperspy.tests.mva.test_decomposition module

hyperspy.tests.mva.test_export module

```python
class hyperspy.tests.mva.test_export.TestMVAExport
    Bases: object
    setup_method(method)
    test_get_bss_factor()
    test_get_bss_loadings()
```

hyperspy.tests.mva.test_onmf module

```python
hyperspy.tests.mva.test_onmf.compare(a, b)
hyperspy.tests.mva.test_onmf.test_corrupted_default()
hyperspy.tests.mva.test_onmf.test_corrupted_robust()
hyperspy.tests.mva.test_onmf.test_default()
hyperspy.tests.mva.test_onmf.test_robust()
```
hyperspy.tests.mva.test_rpca module

Module contents

hyperspy.tests.samfire package

Submodules

hyperspy.tests.samfire.test_goodness_of_fit_tests module

class hyperspy.tests.samfire.test_goodness_of_fit_tests.TestInformationCriteria
    Bases: object
    setup_method(method)
    test_index()
    test_map()

class hyperspy.tests.samfire.test_goodness_of_fit_tests.TestRedChisq
    Bases: object
    setup_method(method)
    test_changing_tolerance()
    test_index()
    test_map()

hyperspy.tests.samfire.test_histogram_segmenter module

class hyperspy.tests.samfire.test_histogram_segmenter.TestHistogramSegmenter
    Bases: object
    setup_method(method)
    test_init()
    test_most_frequent()
    test_update()

hyperspy.tests.samfire.test_red_chisq_weight module

class hyperspy.tests.samfire.test_red_chisq_weight.Test_Red_chisq_weight
    Bases: object
    setup_method(method)
    test_function()
    test_map_noslice()
class hyperspy.tests.samfire.test_strategy.TestGlobalStrategy
    Bases: object
    setup_method(method)
    test_package_values()
    test_refresh_nooverwrite_given()
    test_refresh_nooverwrite_nogiven()
    test_refresh_overwrite_given()
    test_refresh_overwrite_nogiven()
    test_update_marker()

class hyperspy.tests.samfire.test_strategy.TestLocalSimple
    Bases: object
    setup_method(method)
    test_clean()
    test_default_init()
    test_get_distance_array()
    test_radii()
    test_refresh_nooverwrite()
    test_refresh_overwrite()
    test_samf_weight_setters()
    test_update_marker()

class hyperspy.tests.samfire.test_strategy.TestLocalWithModel
    Bases: object
    setup_method(method)
    test_values()

hyperspy.tests.samfire.test_strategy.compare_two_value_dicts(ans_r, ans)

hyperspy.tests.samfire.test_strategy.create_artificial_samfire(shape)

class hyperspy.tests.samfire.test_strategy.someweight
    Bases: object
    function(ind)
    map(calc_pixels, slices=slice(None, None, None))
setup_method(method)
test_append()
test_extend()
test_remove_int()
test_remove_object()

hyperspy.tests.samfire.test_utils module

class hyperspy.tests.samfire.test_utils.TestSamfireUtils
    Bases: object
    setup_method(method)
    test_nearest_indices_cent1()
    test_nearest_indices_cent2()
    test_nearest_indices_cent3()
    test_nearest_indices_cent4()
    test_nearest_indices_ind1()
    test_nearest_indices_ind2()
    test_nearest_indices_ind3()
    test_nearest_indices_ind4()
    test_nearest_indices_ind5()

Module contents

hyperspy.tests.signal package

Submodules

hyperspy.tests.signal.test_1D_tools module

hyperspy.tests.signal.test_2D_tools module

hyperspy.tests.signal.test_apodization module

hyperspy.tests.signal.test_assign_subclass module

class hyperspy.tests.signal.test_assign_subclass.TestConvertBaseSignal
    Bases: object
    setup_method(method)
    test_base_to_1d()
    test_base_to_2d()
    test_base_to_complex()
test_base_to_lazy()
class hyperspy.tests.signal.test_assign_subclass.TestConvertComplexSignal
    Bases: object
    setup_method(method)
    test_complex_to_complex1d()
    test_complex_to_complex2d()
class hyperspy.tests.signal.test_assign_subclass.TestConvertComplexSignal1D
    Bases: object
    setup_method(method)
    test_complex_to_dielectric_function()
class hyperspy.tests.signal.test_assign_subclass.TestConvertSignal1D
    Bases: object
    setup_method(method)
    test_lazy_to_eels_and_back()
    test_signal1d_to_eds_sem()
    test_signal1d_to_eds_tem()
    test_signal1d_to_eels()

test_assignment_class()
class hyperspy.tests.signal.test_assign_subclass.testcase(dtype, sig_dim, sig_type, cls)
    Bases: tuple
    Create new instance of testcase(dtype, sig_dim, sig_type, cls)
    property cls
        Alias for field number 3
    property dtype
        Alias for field number 0
    property sig_dim
        Alias for field number 1
    property sig_type
        Alias for field number 2

test_model_binned module

class hyperspy.tests.signal.test_binned.TestModelBinned
    Bases: object
    setup_method(method)
    test_binned()
    test_unbinned()
    test_dielectric_function_binned_default()
    test_eds_sem_binned_default()
hyperspy.tests.signal.test_binned.test_eds_tem_binned_default()
hyperspy.tests.signal.test_binned.test_eels_spectrum_binned_default()
hyperspy.tests.signal.test_binned.test_image_binned_default()
hyperspy.tests.signal.test_binned.test_signal_binned_default()
hyperspy.tests.signal.test_binned.test_spectrum_binned_default()

hyperspy.tests.signal.test_complex_signal module

hyperspy.tests.signal.test_complex_signal2d module

hyperspy.tests.signal.test_eds_sem module

class hyperspy.tests.signal.test_eds_sem.Test_energy_units
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_beam_energy()
    test_lazy_beam_energy()
    test_lazy_line_energy()
    test_line_energy()

class hyperspy.tests.signal.test_eds_sem.Test_get_lines_intentisity
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test()
    test_background_subraction()
    test_eV()
    test_estimate_integration_windows()
    test_lazy()
    test_lazy_background_subraction()
    test_lazy_eV()
    test_lazy_estimate_integration_windows()
    test_lazy_with_signals_examples()
    test_with_signals_examples()

class hyperspy.tests.signal.test_eds_sem.Test_metadata
    Bases: object

3.1. hyperspy package
lazify(**kwargs)
setup_method(method)
test_SEM_to_TEM()
test_add_elements()
test_add_lines()
test_add_lines_auto()
test_default_param()
test_get_calibration_from()
test_lazy_SEM_to_TEM()
test_lazy_add_elements()
test_lazy_add_lines()
test_lazy_add_lines_auto()
test_lazy_default_param()
test_lazy_get_calibration_from()
test_lazy_rebin_live_time()
test_lazy_sum_live_time()
test_lazy_sum_live_time2()
test_lazy_sum_live_time_out_arg()
test_lazy_take_off_angle()
test_rebin_live_time()
test_sum_live_time()
test_sum_live_time2()
test_sum_live_time_out_arg()
test_take_off_angle()

class hyperspy.tests.signal.test_eds_sem.Test_tools_bulk
    Bases: object
        lazify(**kwargs)
        setup_method(method)
        test_electron_range()
        test_lazy_electron_range()
        test_xray_range()
        test_lazy_xray_range()

hyperspy.tests.signal.test_eds_tem module

class hyperspy.tests.signal.test_eds_tem.Test_eds_markers
    Bases: object
setup_method(method)
test_manual_add_line()
test_manual_remove_element()
test_plot_auto_add()

class hyperspy.tests.signal.test_eds_tem.Test_metadata
    Bases: object
    lazify(**kwargs)
    setup_method(method)
test_TEM_to_SEM()
test_add_elements()
test_are_microscope_parameters_missing()
test_default_param()
test_get_calibration_from()
test_lazy_TEM_to_SEM()
test_lazy_add_elements()
test_lazy_are_microscope_parameters_missing()
test_lazy_default_param()
test_lazy_get_calibration_from()
test_lazy_offset_after_rebin()
test_lazy_rebin_live_time()
test_lazy_sum_live_time1()
test_lazy_sum_live_time2()
test_lazy_sum_live_time_out_arg()
test_lazy_sum_minimum_missing()
test_offset_after_rebin()
test_rebin_live_time()
test_sum_live_time1()
test_sum_live_time2()
test_sum_live_time_out_arg()
test_sum_minimum_missing()

class hyperspy.tests.signal.test_eds_tem.Test_quantification
    Bases: object
    lazify(**kwargs)
    setup_method(method)
test_edx_cross_section_to_zeta()
test_lazy_edx_cross_section_to_zeta()
test_lazy_metadata()
test_lazy_quant_cross_section()

class hyperspy.tests.signal.test_eds_tem.Test_simple_model
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_intensity()
    test_intensity_dtype_uint()
    test_lazy_intensity()
    test_lazy_intensity_dtype_uint()

class hyperspy.tests.signal.test_eds_tem.Test_vacum_mask
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_vacuum_mask()
    test_vacuum_mask()

hyperspy.tests.signal.test_eds_tem.test_with_signals_examples()
```
lazify(**kwargs)

setup_method(method)

test_find_peaks1D_ohaver_high_amp_thres()
test_find_peaks1D_ohaver_zero_value_bug()
test_lazy_find_peaks1D_ohaver_high_amp_thres()
test_lazy_find_peaks1D_ohaver_zero_value_bug()
```

definition

```
hyperspy.tests.signal.test_folding module

class hyperspy.tests.signal.test_folding.TestSignalFolding
    Bases: object

    lazify(**kwargs)

    setup_method(method)

    test_lazy_unfold_full_by_keyword()
    test_lazy_unfold_navigation()
    test_lazy_unfold_navigation_by_keyword()
    test_lazy_unfold_nothing_by_keyword()
    test_lazy_unfold_signal()
    test_lazy_unfold_signal_by_keyword()
    test_lazy_unfolded_context_manager()
    test_lazy_unfolded_full_by_keywords()
    test_lazy_unfolded_navigation_by_keyword()
    test_lazy_unfolded_nothin_by_keyword()
    test_lazy_unfolded_repr()
    test_lazy_unfolded_signal_by_keyword()
    test_unfold_full_by_keyword()
    test_unfold_navigation()
    test_unfold_navigation_by_keyword()
    test_unfold_nothing_by_keyword()
    test_unfold_signal()
    test_unfold_signal_by_keyword()
    test_unfolded_context_manager()
    test_unfolded_full_by_keywords()
    test_unfolded_navigation_by_keyword()
    test_unfolded_nothin_by_keyword()
    test_unfolded_repr()
    test_unfolded_signal_by_keyword()
```

3.1. hyperspy package
class hyperspy.tests.signal.test_folding.TestSignalVarianceFolding
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_unfold_navigation()
    test_lazy_unfold_signal()
    test_unfold_navigation()
    test_unfold_signal()

hyperspy.tests.signal.test_fourier_transform module

hyperspy.tests.signal.test_hologram_image module

hyperspy.tests.signal.test_image module

class hyperspy.tests.signal.test_image.Test2D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_signal1D()
    test_to_signal1D()

class hyperspy.tests.signal.test_image.Test3D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_signal1D()
    test_to_signal1D()

class hyperspy.tests.signal.test_image.Test4D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_image()
    test_to_image()

hyperspy.tests.signal.test_inheritance module

hyperspy.tests.signal.test_kramers_kronig_transform module

hyperspy.tests.signal.test_lazy module
hyperspy.tests.signal.test_map_method module

hyperspy.tests.signal.test_remove_background module

hyperspy.tests.signal.test_rgb module

hyperspy.tests.signal.test_signal_operators module

hyperspy.tests.signal.test_signal_subclass_conversion module

hyperspy.tests.signal.test_spectrum module

class hyperspy.tests.signal.test_spectrum.Test2D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_signal2D()
    test_to_signal2D()

class hyperspy.tests.signal.test_spectrum.Test3D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_signal2D()
    test_to_signal2D()

class hyperspy.tests.signal.test_spectrum.Test4D
    Bases: object
    lazify(**kwargs)
    setup_method(method)
    test_lazy_to_signal2D()
    test_to_signal2D()

hyperspy.tests.signal.test_tools module

Module contents

hyperspy.tests.utils package

Submodules

hyperspy.tests.utils.test_attrsetter module
hyperspy.tests.utils.test_eds module

hyperspy.tests.utils.test_eds.test_xray_lines_near_energy()

hyperspy.tests.utils.test_material module

class hyperspy.tests.utils.test_material.TestWeightToFromAtomic
    Bases: object
    setup_method(method)
    test_atomic_to_weight()
    test_multi_dim()
    test_weight_to_atomic()

hyperspy.tests.utils.test_material.test_density_of_mixture()

hyperspy.tests.utils.test_material.test_mac()

hyperspy.tests.utils.test_material.test_mixture_mac()

hyperspy.tests.utils.test_print_known_signal_types module

class hyperspy.tests.utils.test_print_known_signal_types.TestPrintKnownSignalTypes
    Bases: object
    test_html_output()
    test_text_output()

hyperspy.tests.utils.test_roi module

hyperspy.tests.utils.test_slugify module

hyperspy.tests.utils.test_slugify.test_slugify()

hyperspy.tests.utils.test_stack module

class hyperspy.tests.utils.test_stack.TestToolsUtilsStack
    Bases: object
    setup_method(method)
    test_stack_bigger_than_ten()
    test_stack_broadcast_number()
    test_stack_broadcast_number_not_default()
    test_stack_default()
    test_stack_not_default()
    test_stack_of_stack()
Module contents

Submodules

hyperspy.tests.test_dictionary_tree_browser module

class hyperspy.tests.test_dictionary_tree_browser.TestDictionaryBrowser
    Bases: object
    setup_method(method)
    test_add_dictionary()
    test_add_signal_in_dictionary()
    test_date_time_nanosecond_precision()
    test_date_time_now()
    test_get_item()
    test_has_item()
    test_signal_to_dictionary()

hyperspy.tests.test_events module

hyperspy.tests.test_interactive module

Module contents

hyperspy.utils package

Submodules

hyperspy.utils.eds module

Markers that can be added to Signal plots.

Example

```python
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> m = hs.plot.markers.rectangle(x1=150, y1=100, x2=400, y2=400, color='red')
>>> im.add_marker(m)
```

hyperspy.utils.material module

3.1. hyperspy package 455
**hyperspy.utils.model module**

Model functions.

The **model** module contains the following submodules:

- **components1D**  1D components for HyperSpy model.
- **components2D**  2D components for HyperSpy model.

**hyperspy.utils.model_selection module**

- **hyperspy.utils.model_selection.AIC**(model)
  Calculates the Akaike information criterion
  \[ AIC = 2k - 2 \ln(L) \]
  where \( L \) is the maximum likelihood function value, \( k \) is the number of free parameters.

- **hyperspy.utils.model_selection.AICc**(model)

- **hyperspy.utils.model_selection.BIC**(model)
  Calculates the Bayesian information criterion
  \[ BIC = -2 \ln(L) + k \ln(n) \]
  where \( L \) is the maximum likelihood function, \( k \) is the number of free parameters, and \( n \) is the number of data points (observations) / sample size.

**hyperspy.utils.parallel_pool module**

**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

  - **setup()**
    sets up the requested pool

  - **sleep()**
    sleeps for the requested (or timeout) time

- **has_pool**
  Boolean if the pool is available and active.

  - **Type:** bool

- **pool**
  The pool object.

  - **Type:** (ipyparallel.load_balanced_view, multiprocessing.Pool)

- **ipython_kwargs**
  The dictionary with Ipyparallel connection arguments.

  - **Type:** dict
timeout
Timeout for either pool when waiting for results.
   Type  float

num_workers
The number of workers actually created (may be less than requested, but can’t be more).
   Type  int

timestep
Can be used as “ticks” to adjust CPU load when building upon this class.
   Type  float

is_ipyparallel
If the pool is ipyparallel-based
   Type  bool

is_multiprocessing
If the pool is multiprocessing-based
   Type  bool

Creates the ParallelPool and sets it up.

Parameters

• **num_workers** ({None, int}) – the (max) number of workers to create. If less are available, smaller number is actually created.

• **ipyparallel** ({None, bool}) – which pool to set up. True - ipyparallel. False - multiprocessing. None - try ipyparallel, then multiprocessing if failed.

• **ipython_kwargs** ({None, dict}) – arguments that will be passed to the ipyparallel.Client when creating. Not None implies ipyparallel=True.

property has_pool
Returns bool if the pool is ready and set-up

property is_ipyparallel
Returns bool if the pool is ipyparallel-based

property is_multiprocessing
Returns bool if the pool is multiprocessing-based

setup(ipyparallel=None)
Sets up the pool.

Parameters ipyparallel ({None, bool}) – 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, float}) – How long the pool should sleep for in seconds. If None (default), sleeps for “timestep”

property timestep
hyperspy.utils.plot module

Plotting functions.

Functions:

- **plot_spectra**, **plot_images**  Plot multiple spectra/images 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.

The `plot` module contains the following submodules:

- **markers**  Markers that can be added to `Signal` plots.

hyperspy.utils.roi module

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.

hyperspy.utils.samfire module

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 local strategies to use in SAMFire
- **SamfirePool**  The parallel pool, customized to run SAMFire.

Module contents

Functions that operate on `Signal` instances and other goodies.

- **stack**  Stack `Signal` instances.

Subpackages:

- **material**  Tools related to the material under study.
- **plot**  Tools for plotting.
eds  Tools for energy-dispersive X-ray data analysis.

example_signals  A few example of signal

hyperspy.utils.print_known_signal_types()
  Print all known signal_type's

This includes signal_type's from all installed packages that extend HyperSpy.

Examples

```
>>> hs.print_known_signal_types()
+--------------------+---------------------+--------------------+----------+
| signal_type | aliases | class name | package |
+--------------------+---------------------+--------------------+----------+
| DielectricFunction | dielectric function | DielectricFunction | hyperspy |
| EDS_SEM | | EDSSEMSpectrum | hyperspy |
| EDS_TEM | | EDSTEMSpectrum | hyperspy |
| EELS | TEM EELS | EELSSpectrum | hyperspy |
| hologram | | HologramImage | hyperspy |
| MySignal | | MySignal | hspy_ext |
+--------------------+---------------------+--------------------+----------+
```

3.1.2 Submodules

hyperspy.Release module

hyperspy._lazy_signals module

hyperspy.api module

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 has been imported as hs, numpy as np and matplotlib.pyplot as plt.

Functions:

- **create_model**  Create a model for curve fitting.
- **get_configuration_directory_path**  Return the configuration directory path.
- **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 exeecting its gui method i.e. preferences.gui().
- **stack**  Stack several signals.
- **interactive**  Define operations that are automatically recomputed on event changes.
set_log_level  Conveniece function to set HyperSpy’s the log level.

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**  Contains the components module with components that can be used to create a model for curve fitting.

**eds**  Functions for energy dispersive X-rays data analysis.

**material**  Useful functions for materials properties and elements database that includes physical properties and X-rays and EELS energies.

**plot**  Plotting functions that operate on multiple signals.

**datasets**  Example 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 doctrings.

**hyperspy.api_nogui module**

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:

```python
>>> 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 has been imported as hs, numpy as np and matplotlib.pyplot as plt.

Functions:

**create_model**  Create a model for curve fitting.

**get_configuration_directory_path**  Return the configuration directory path.

**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 exeacting its gui method i.e. preferences.gui().

**stack**  Stack several signals.

**interactive**  Define operations that are automatically recomputed on event changes.

**set_log_level**  Conveniece function to set HyperSpy’s the log level.

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 Contains the components module with components that can be used to create a model for curve fitting.

eds Functions for energy dispersive X-rays data analysis.

material Useful functions for materials properties and elements database that includes physical properties and X-rays and EELS energies.

plot Plotting functions that operate on multiple signals.

datasets Example 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 doctrings.

hyperspy.api_nogui.get_configuration_directory_path()

hyperspy.axes module

class hyperspy.axes.AxesManager (axes_list)
    Bases: traits.has_traits.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 support 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.

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.

    Type  tuple

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.

    Type  tuple

signal_axes, navigation_axes
    Contain the corresponding DataAxis objects

    Type  list

Examples

Create a spectrum with random data

```python
>>> 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 | 1 | <undefined>

```python
g499 = s.axes_manager[0]
g500 = s.axes_manager[3]
g501 = s.axes_manager[1]
g502 = s.axes_manager[2]
g503 = s.axes_manager[1].name = "y"
g504 = s.axes_manager["y"]
g505 = 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)  
(0, 1, 0) (0, 1, 0)  
(1, 1, 0) (1, 1, 0)  
(2, 1, 0) (2, 1, 0)  
(3, 1, 0) (3, 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)  
(0, 0, 1) (0, 0, 1)  
(1, 0, 1) (1, 0, 1)  
(2, 0, 1) (2, 0, 1)  
(3, 0, 1) (3, 0, 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)  
(0, 2, 1) (0, 2, 1)  
(1, 2, 1) (1, 2, 1)  
(2, 2, 1) (2, 2, 1)  
(3, 2, 1) (3, 2, 1)
```

as_dictionary()  

**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  

Return type 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 | string | iterable of DataAxis | 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 string of the same length than axes | str | None]) – Default = None If list, the selected axes will be converted to the provided units. If str, 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 units 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.

**property coordinates**

Get the coordinates of the navigation axes.

**Returns**

**Return type** list

**copy()**

**create_axes**(axes_list)

Given a list of dictionaries defining the axes properties create the DataAxis instances and add them to the AxesManager.

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.

**See also:**

__append_axis()

**deepcopy()**

**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 strings 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.

**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 the index of the navigation axes.

Returns
- **Return type** `list`

**key_navigator** *(event)*
Set hotkeys for controlling the indices of the navigator plot

**property navigation_extent**

**property navigation_indices_in_array**

**remove** *(axes)*
Remove one or more axes

**set_signal_dimension** *(value)*
Set the dimension of the signal.

- **value**
  - **Type** `int`

  **Raises**
  - • `ValueError` if value if greater than the number of axes or –
  - • is negative –

**property shape**

**show**()

**property signal_extent**

**property signal_indices_in_array**

**update_axes_attributes_from** *(axes, attributes=['scale', 'offset', 'units'])*
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 `DataAxis` instances.)* – The axes to copy the attributes from.
  - • **attributes** *(iterable of strings.)* – The attributes to copy.

**class hyperspy.axes.DataAxis** *(size, index_in_array=None, name=<undefined>, scale=1.0, offset=0.0, units=<undefined>, navigate=<undefined>)*

- **Bases:** `traits.has_traits.HasTraits`, `hyperspy.axes.UnitConversion`

**calibrate** *(value_tuple, index_tuple, modify_calibration=True)*

**copy**()

**get_axis_dictionary**()

**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 strings 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.

**index2value** *(index)*

**property index_in_array**

**property index_in_axes_manager**

**property offset_as_quantity**

**property scale_as_quantity**

**update_axis** *

**update_from** *(axis, attributes=[‘scale’, ‘offset’, ‘units’])*

Copy values of specified axes fields from the passed AxesManager.

Parameters

- **axis** *(DataAxis)* – The DataAxis instance to use as a source for values.

- **attributes** *(iterable container of strings.)* – The name of the attribute to update. If the attribute does not exist in either of the AxesManagers, an AttributeError will be raised.

Returns

**Return type** A boolean indicating whether any changes were made.

**update_index_bounds** *

**value2index** *(value, rounding=<built-in function round>)*

Return the closest index to the given value if between the limit.

Parameters **value** *(number or numpy array)* –

Returns index

**Return type** integer or numpy array

**Raises** ValueError if any value is out of the axis limits.

**value_range_to_indices** *(vl, v2)*

Convert the given range to index range.

When an out of the axis limits, the endpoint is used instead.

Parameters **v2** *(vl, )* – The end points of the interval in the axis units. v2 must be greater than vl.

**class hyperspy.axes.UnitConversion** *(units=<undefined>, scale=1.0, offset=0.0)*

Bases: object

**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
HyperSpy Documentation, Release 1.5.1.dev

- **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.

**property units**

hyperspy.axes.generate_axis *(offset, scale, size, offset_index=0)*

Creates an axis given the offset, scale and number of channels

Alternatively, the offset_index of the offset channel can be specified.

**Parameters**

- **offset** *(float)* –
- **scale** *(float)* –
- **size** *(number of channels)* –
- **offset_index** *(int)* – offset_index number of the offset

**Returns**

**Return type** Numpy array

**class hyperspy.axes.ndindex_nat (*shape)*

**class hyperspy.component.Component (parameter_name_list)*

**property active_is_multidimensional**

**as_dictionary** *(fullcopy=True)*

Returns component as a dictionary For more information on method and conventions, see hyperspy.miss.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**

dic – A dictionary, containing at least the following fields: parameters : list

- a list of dictionaries of the parameters, one per

- _whitelist [dictionary] a dictionary with keys used as references saved attributes, for more information, see hyperspy.miss.export_dictionary.export_to_dictionary()

- any field from _whitelist.keys() *

**Return type** dictionary
**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_stored_values** *(only_fixed=False)*

**fetch_values_from_array** *(p, p_std=None, onlyfree=False)*

**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 strings 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.

**init_parameters** *(parameter_name_list)*

**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, fancy=True)*

Prints the current values of the component’s parameters. 

**set_parameters_free** *(parameter_name_list=None)*

Sets parameters in a component to free.

**Parameters**

- **parameter_name_list** *(None or list of strings, 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.
Examples

```python
>>> v1 = hs.model.components1D.Voigt()
>>> v1.set_parameters_free()
>>> v1.set_parameters_free(parameter_name_list=['area', 'centre'])
```

See also:

- `set_parameters_not_free()`, `hyperspy.model.BaseModel.set_parameters_not_free()`

**set_parameters_not_free** *(parameter_name_list=None)*

Sets parameters in a component to not free.

Parameters **parameter_name_list** *(None or list of strings, 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.

Examples

```python
>>> v1 = hs.model.components1D.Voigt()
>>> v1.set_parameters_not_free()
>>> v1.set_parameters_not_free(parameter_name_list=['area', 'centre'])
```

See also:

- `set_parameters_free()`, `hyperspy.model.BaseModel.set_parameters_free()`
- `hyperspy.model.BaseModel.set_parameters_not_free()`

**store_current_parameters_in_map** ()

**summary** ()

**update_number_parameters** ()

**class** hyperspy.component.NoneFloat *(default_value=<traits.trait_handlers.NoDefaultSpecified object>, **metadata)*

This constructor method is the only method normally called directly by client code. It defines the trait. The default implementation accepts an optional, untype-checked default value, and caller-supplied trait metadata. Override this method whenever a different method signature or a type-checked default value is needed.

**default_value** = None

**validate** *(object, name, value)*

Validates that a specified value is valid for this trait.

Note: The ‘fast validator’ version performs this check in C.

**class** hyperspy.component.Parameter

**Bases**: traits.has_traits.HasTraits

Model parameter

**value**

The value of the parameter for the current location. The value for other locations is stored in map.

Type **float** or array

**bmin, bmax**

Lower and upper bounds of the parameter value.
Type  float

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

    Type  [None, Parameter]

twin_function_expr
Expression of the twin_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.

    Type  str

twin_inverse_function
Expression of the twin_inverse_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.

    Type  str
	
twin_function
Setting this attribute manually is deprecated in HyperSpy newer than 1.1.2. It will become private
in HyperSpy 2.0. Please use "twin_function_expr" instead.

    Type  function

twin_inverse_function
Setting this attribute manually is deprecated in HyperSpy newer than 1.1.2. It will become private
in HyperSpy 2.0. Please use "twin_inverse_function_expr" instead.

    Type  function

etype_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.

    Type  bool

ete_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.

    Type  bool
	
tas_signal (field = 'values')
Get a parameter map as a signal object

plot ()
Plots the value of the Parameter at all locations.

export (folder=None, name=None, format=None, save_std=False)
Saves the value of the parameter map to the specified format

connect, disconnect (function)
Call the functions connected when the value attribute changes.
as_dictionary (fullcopy=True)
   Returns parameter as a dictionary, saving all attributes from self._whitelist.keys() For more information see 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
   dic –
   _id_name [string] _id_name of the original parameter, used to create the dictionary. Has to match with the self._id_name
   _twins [list] a list of ids of the twins of the parameter
   _whitelist [dictionary] a dictionary, which keys are used as keywords to match with the parameter attributes. For more information see hyperspy.misc.export_dictionary.export_to_dictionary()

   • any field from _whitelist.keys() *

   Return type dictionary with the following 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') –
   :raises NavigationDimensionError : if the navigation dimension is 0:

assign_current_value_to_all (mask=None)
   Assign the current value attribute to all the indices

   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 extbounded**

**property extforcepositive**

**fetch()**

Fetch the stored value and std attributes.

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 strings 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.

**plot** *(**kwargs)**

Plot parameter signal.

**Parameters** **kwargs – Any extra keyword arguments are passed to the signal plot.

**Example**

```
>>> parameter.plot() #doctest: +SKIP
```

Set the minimum and maximum displayed values

```
>>> parameter.plot(vmin=0, vmax=1) #doctest: +SKIP
```

**store_current_value_in_array()**

Store the value and std attributes.

See also:

```
fetch(), assign_current_value_to_all()
```

**property twin**

```
twin_function = None
```

**property twinfunctionexpr**

**property twininversefunction**

**property twininversefunctionexpr**

---

**hyperspy.components1d module**

Components that can be used to define a 1D model for e.g. curve fitting.

There are some components that are only useful for one particular kind of signal and therefore their name are preceded by the signal name: eg. eels_el_edge.
Writing a new template is really easy, just edit _template.py and maybe take a look to the other components.

For more details see each component docstring.

Arctan Arctan function component... Bleasdale Bleasdale function component... DoublePowerLaw Double power law component for EELS spectra... EELSCLEdge EELS core loss ionisation edge from hydroge... 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, wit... HeavisideStep The Heaviside step function... Logistic Logistic function (sigmoid or s-shaped curv... Lorentzian Cauchy-Lorentz distribution (a.k.a. Lorentz... Offset Component to add a constant value in the y-... Polynomial n-order polynomial component. (DEPRECATED)... PowerLaw Power law component... RC... SEE Secondary electron emission component for P... ScalableFixedPattern Fixed pattern component with interpolation... SkewNormal Skew normal distribution component... Vignetting... Voigt Voigt profile component with support for sh... VolumePlasmonDrude Drude volume plasmon energy loss function c...
class hyperspy.defaults_parser.GUIs
Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.GeneralConfig
Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.PlotConfig
Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.Preferences
Bases: traits.has_traits.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 strings 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.

save()

hyperspy.defaults_parser.config2template(template, config)

hyperspy.defaults_parser.dictionary_from_template(template)

hyperspy.defaults_parser.file_version(fname)

hyperspy.defaults_parser.guess_gos_path()

hyperspy.defaults_parser.template2config(template, config)

hyperspy.events module

class hyperspy.events.Event (doc=", arguments=None)
Bases: object

Create an Event object.

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.

>>> 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):
(continues on next page)
property arguments

connect (function, kwargs=‘all’)

Connects a function to the event. Arguments: function : callable

The function to call when the event triggers.

kwargs [(tuple or list, dictionary, ‘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. :param function: :type function: function :param return_connection_kwargs: 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.

>>> 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()

See also:

suppress_callback(), Events.suppress()

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.
>>> 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()

See also:
suppress(), Events.suppress()

trigger(**kwars)
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 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

```python
>>> 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
supress()  
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.

supress()  
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.

```python
>>> 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()
```

See also:

Event.suppress(), Event.suppress_callback()

hyperspy.exceptions module

exception hyperspy.exceptions.ByteOrderError(order=")"
Bases: Exception

exception hyperspy.exceptions.DM3DataTypeError(value="
Bases: Exception

exception hyperspy.exceptions.DM3FileVersionError(value="
Bases: Exception

exception hyperspy.exceptions.DM3TagError(value="
Bases: Exception

exception hyperspy.exceptions.DM3TagIDError(value="
Bases: Exception

exception hyperspy.exceptions.DM3TagTypeError(value="
Bases: Exception

exception hyperspy.exceptions.DataDimensionError(msg)
Bases: Exception

exception hyperspy.exceptions.ImageIDError(value="
Bases: Exception

exception hyperspy.exceptions.ImageModeError(value="
Bases: Exception

exception hyperspy.exceptions.MissingParametersError(parameters)
Bases: Exception
exception hyperspy.exceptions.NavigationDimensionError (navigation_dimension, expected_navigation_dimension)
    Bases: Exception

exception hyperspy.exceptions.NavigationSizeError (navigation_size, expected_navigation_size)
    Bases: Exception

exception hyperspy.exceptions.NoInteractiveError
    Bases: Exception

exception hyperspy.exceptions.ShapeError (value)
    Bases: Exception

exception hyperspy.exceptions.SignalDimensionError (output_dimension, expected_output_dimension)
    Bases: Exception

exception hyperspy.exceptions.SignalSizeError (signal_size, expected_signal_size)
    Bases: Exception

exception hyperspy.exceptions.VisibleDeprecationWarning
    Bases: UserWarning

    Visible deprecation warning. By default, python will not show deprecation warnings, so this class provides a visible one.

exception hyperspy.exceptions.WrongObjectError (is_str, must_be_str)
    Bases: Exception

hyperspy.extensions module

hyperspy.interactive module

class hyperspy.interactive.Interactive (f, event='auto', recompute_out_event='auto', *args, **kwargs)
    Bases: object

    Chainable operations on Signals that update on events.
    Update operation result when a given event is triggered.

    Parameters

    • \(f\) (function or method) – A function that returns an object and that optionally can place the result in an object given through the \(out\) keyword.

    • \(event\) (\(\text{Event, }{"auto"}, \text{None, iterable of events}\)) – 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.

    • \(recompute\_out\_event\) (\(\text{Event, }{"auto"}, \text{None, iterable of events}\)) – 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 \(Signal\) 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, **kwargs Arguments and keyword arguments to be passed to f.

recompute_out()
update()

hyperspy.interactive.interactive(f, event='auto', recompute_out_event='auto', *args, **kwargs)

Update operation result when a given event is triggered.

Parameters

- **f** (function or method) – A function that returns an object and that optionally can place the result in an object given through the out keyword.
- **event** ({Event, "auto", None, iterable of events}) – 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** ({Event, "auto", None, iterable of events}) – 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 Signal 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, **kwargs Arguments and keyword arguments to be passed to f.

hyperspy.io module

class hyperspy.io.assign_signal_subclass(dtype, signal_dimension, signal_type=",", lazy=False)

Given record_by and signal_type return the matching Signal subclass.

Parameters

- **dtype** (dtype) –
- **signal_dimension** (int) –
- **signal_type** ("EELS", "EDS", "EDS_SEM", "EDS_TEM", "DielectricFunction", ",", str) –
- **lazy** (bool) –

Returns

Return type Signal or subclass

class hyperspy.io.dict2signal(signal_dict, lazy=False)

Create a signal (or subclass) instance defined by a dictionary

Parameters **signal_dict** (dictionary) –

Returns **s**

Return type Signal or subclass
Load potentially multiple supported file into an hyperspy structure.

Supported formats: hspy (HDF5), msa, Gatan dm3, Ripple (rpl+raw), Bruker bcf and spx, FEI ser and emi, SEMPER unf, EMD, EDAX spd/spc, tif, and a number of image formats.

Any extra keyword is passed to the corresponding reader. For available options see their individual documentation.

Parameters

- `filenames`: The filename to be loaded. If None, a window will open to select a file to load. If a valid filename is passed in that single file is loaded. If multiple file names are passed in a list, a list of objects or a single object containing the data of the individual files stacked are returned. This behaviour is controlled by the `stack` parameter (see bellow). Multiple files can be loaded by using simple shell-style wildcards, e.g. `my_file*.msa` loads all the files that starts by `my_file` and has the `.msa` extension.
- `signal_type`: The acronym that identifies the signal type. The value provided may determine the Signal subclass assigned to the data. If None the value is read/guessed from the file. Any other value overrides the value stored in the file if any. For electron energy-loss spectroscopy use “EELS”. For energy dispersive x-rays use “EDS_TEM” if acquired from an electron-transparent sample — as it is usually the case in a transmission electron microscope (TEM) —, “EDS_SEM” if acquired from a non electron-transparent sample — as it is usually the case in a scanning electron microscope (SEM). If “” (empty string) the value is not read from the file and is considered undefined.
- `stack`: If True and multiple filenames are passed in, 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`: 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`.
- `new_axis_name`: 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`: Open the data lazily - i.e. without actually reading the data from the disk until required. Allows opening arbitrary-sized datasets. The default is `False`.
- `convert_units`: If True, convert the units using the `convert_to_units` method of the `axes_manager`. If False, does nothing. The default is `False`.
- `print_info`: For SEMPER unf- and EMD (Berkeley)-files, if True (default is False) additional information read during loading is printed for a quick overview.
- `downsample`: 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`: For Bruker bcf files, if set to numerical (default is None) bcf is parsed into array with depth cutoff at corresponding given en-
ergy. This allows to conserve the memory, with cutting-off unused spectra’s tail, or force enlargement of the spectra size.

- **select_type** ("spectrum_image", 'image', 'single_spectrum', None) – 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 return single spectra either only the spectrum image or only the images (including EDS map for Velox emd files) or only the single spectra (for Velox emd files).

- **first_frame** (int (default 0)) – Only for Velox emd files: load only the data acquired after the specified frame.

- **last_frame** (None or int (default None)) – Only for Velox emd files: load only the data acquired up to specified frame. If None, load up the data to the end.

- **sum_frames** (bool (default is True)) – Only for Velox emd files: if False, load each EDS frame individually.

- **sum_EDS_detectors** (bool (default is True)) – Only for Velox emd files: if True, the signal from the different detector are summed. If False, a distinct signal is returned for each EDS detectors.

- **rebin_energy** (int, a multiple of the length of the energy dimension (default 1)) – Only for Velox emd files: rebin the energy axis by the integer provided during loading in order to save memory space.

- **SI_dtype** (numpy.dtype) – 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 (default False)) – Only for Velox emd files: if True, load the stack of STEM images acquired simultaneously as the EDS spectrum image.

- **dataset_name** (string or list, 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) files.

- **only_valid_data** (bool, optional) – Only for FEI emi/ser file 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**

- **Return type** Signal instance or list of signal instances

**Examples**

Loading a single file providing the signal type:

```python
>>> d = hs.load('file.dm3', signal_type="EDS_TEM")
```

Loading multiple files:

```python
>>> d = hs.load('file1.dm3','file2.dm3')
```

Loading multiple files matching the pattern:
>>> d = hs.load('file*.dm3')

Loading (potentially larger than the available memory) files lazily and stacking:

>>> s = hs.load('file*.blo', lazy=True, stack=True)

_hyperspy.io_.load_single_file(filename, **kwds)

Load any supported file into an HyperSpy structure. Supported formats: netCDF, msa, Gatan dm3, Ripple (rpl+raw), Bruker bcf, FEI ser and emi, EDAX spc and spd, hspy (HDF5), and SEMPER unf.

**Parameters**

- **filename** (string) – File name (including the extension)

_hyperspy.io_.load_with_reader(filename, reader, signal_type=None, convert_units=False, **kwds)

_hyperspy.io_.save(filename, signal, overwrite=None, **kwds)

**hyperspy.logger module**

_hyperspy.logger_.set_log_level(level)

Convenience function to set the log level of all hyperspy modules.

**Parameters**

- **level** ((int | 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'

For normal logging of hyperspy functions, you can set the log level like this:

```python
>>> import hyperspy.api as hs
>>> hs.set_log_level('INFO')
>>> hs.load(r'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)>
```

If you need the log output during the initial import of hyperspy, you should set the log level like this:

```python
>>> from hyperspy.logger import set_log_level
>>> 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.model module**

```python
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 components that are added to the model using `append()` or `extend()`. There are many predefined components available in the `components` module. 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.

    signal
        It contains the data to fit.
        
        **Type** BaseSignal instance

    chisq
        Chi-squared of the signal (or np.nan if not yet fit)
        
        **Type** A BaseSignal of floats

    dof
        Degrees of freedom of the signal (0 if not yet fit)
        
        **Type** A BaseSignal of integers

    red_chisq
        Reduced chi-squared.
        
        **Type** BaseSignal instance

    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.

        **Type** ModelComponents instance

    append()
        Append one component to the model.

    extend()
        Append multiple components to the model.

    remove()
        Remove component from model.

    as_signal()
        Generate a BaseSignal instance (possible multidimensional) from the model.

    store_current_values()
        Store the value of the parameters at the current position.

    fetch_stored_values()
        Fetch stored values of the parameters.
```

update_plot()
Force a plot update. (In most cases the plot should update automatically.)

set_signal_range, remove_signal_range, reset_signal_range,
add_signal_range.
Customize the signal range to fit.

fit, multifit
Fit the model to the data at the current position or the full dataset.

save_parameters2file, load_parameters_from_file
Save/load the parameter values to/from a file.

plot()
Plot the model and the data.

enable_plot_components, disable_plot_components
Plot each component separately. (Use after plot.)

set_current_values_to()
Set the current value of all the parameters of the given component as the value for all the dataset.

export_results()
Save the value of all the parameters in separate files.

plot_results()
Plot the value of all parameters at all positions.

print_current_values()
Print the value of the parameters at the current position.

enable_adjust_position, disable_adjust_position
Enable/disable interactive adjustment of the position of the components that have a well defined position. (Use after plot.)

fit_component()
Fit just the given component in the given signal range, that can be set interactively.

set_parameters_not_free, set_parameters_free
Fit the free status of several components and parameters at once.

set_parameters_value()
Set the value of a parameter in components in a model to a specified value.

as_dictionary()
Exports the model to a dictionary that can be saved in a file.

See also:
Model1D, Model2D

append(thing)
Add component to Model.

Parameters thing (Component instance) –

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** *(a complete dictionary of the model, which includes at least the following fields –)*
  - **components** *(list)* a list of dictionaries of components, one per
  - **_whitelist** *(dictionary)* a dictionary with keys used as references for saved attributes, for more information, see `hyperspy.misc.export_dictionary.export_to_dictionary()`
  - any field from `_whitelist`.keys() *

**Examples**

```python
>>> s = signals.Signal1D(np.random.random((10,100)))
>>> m = s.create_model()
>>> l1 = components1D.Lorentzian()
>>> l2 = 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, parallel=None)*

Returns a recreation of the dataset using the model. The spectral range that is not fitted is filled with nans.

**Parameters**

- **component_list** *(list of HyperSpy components, 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 spectral range that is not fitted 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, 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.
- **parallel** *(None, bool, or int)* – If True, perform computation in parallel using multiple cores. If int, use as many cores as specified. If None the default from the preferences settings is used.

**Returns** *BaseSignal*

**Return type** An instance of the same class as BaseSignal.

**Examples**

```python
>>> 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)
(continues on next page)```
>>> m.append(l2)
>>> s1 = m.as_signal()
>>> s2 = m.as_signal(component_list=[l1])

**assign_current_values_to_all** (*components_list=*, *mask=*)

Set parameter values for all positions to the current ones.

**Parameters**

- **component_list** (*list of components, 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.

- **mask** (*boolean numpy array or None, optional*) – The operation won’t be performed where mask is True.

**create_samfire** (*workers=*, *setup=True, **kwargs*)

Creates a SAMFire object.

**Parameters**

- **workers** (*{None, 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**

**enable_plot_components**

**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=*, *format=*, *save_std=*, *only_free=*, *only_active=*)

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)
Extend list by appending elements from the iterable.

fetch_stored_values(only_fixed=False)
Fetch the value of the parameters that has been previously stored.

Parameters

only_fixed (bool, optional) – If True, only the fixed parameters are fetched.

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.

Parameters

• array (array) – array with the parameter values
• array_std (None, array) – array with the standard deviations of parameters

fit(fitter='leastsq', method='ls', grad=False, bounded=False, ext_bounding=False, update_plot=False, **kwargs)
Fits the model to the experimental data.

The chi-squared, reduced chi-squared and the degrees of freedom are computed automatically when fitting. They are stored as signals, in the chisq, red_chisq and dof. Note that unless metadata.Signal.Noise_properties.variance contains an accurate estimation of the variance of the data, the chi-squared and reduced chi-squared cannot be computed correctly. This is also true for homocedastic noise.

Parameters

• fitter ("leastsq", "mpfit", "odr", "Nelder-Mead") –

The optimization algorithm used to perform the fitting. Default is “leastsq”.

“leastsq” performs least-squares optimization, and supports bounds on parameters.

"mpfit" performs least-squares using the Levenberg–Marquardt algorithm and supports bounds on parameters.

"odr" performs the optimization using the orthogonal distance regression algorithm. It does not support bounds.

"Nelder-Mead”, “Powell”, “CG”, “BFGS”, “Newton-CG”, “L-BFGS-B” and “TNC” are wrappers for scipy.optimize.minimize(). Only “L-BFGS-B” and “TNC” support bounds.

“Differential Evolution” is a global optimization method.

“leastsq”, “mpfit” and “odr” can estimate the standard deviation of the estimated value of the parameters if the “metadata.Signal.Noise_properties.variance” attribute is defined. Note that if it is not defined, the standard deviation is estimated using a variance of 1. If the noise is heteroscedastic, this can result in a biased estimation.
of the parameter values and errors. If `variance` is a `Signal` instance of the same `navigation_dimension` as the signal, and `method` is “ls”, then weighted least squares is performed.

- **method** ('ls', 'ml', 'custom') – Choose ‘ls’ (default) for least-squares and ‘ml’ for Poisson maximum likelihood estimation. The latter is not available when ‘fitter’ is “leastsq”, “odr” or “mpfit”. ‘custom’ allows passing your own minimisation function as a kwarg “min_function”, with optional gradient kwarg “min_function_grad”. See User Guide for details.

- **grad** (bool) – If True, the analytical gradient is used if defined to speed up the optimization.

- **bounded** (bool) – If True performs bounded optimization if the fitter supports it.

- **update_plot** (bool) – If True, the plot is updated during the optimization process. It slows down the optimization but it permits to visualize the optimization progress.

- **ext_bounding** (bool) – If True, enforce bounding by keeping the value of the parameters constant out of the defined bounding area.

- ****kwargs** (key word arguments) – Any extra key word argument will be passed to the chosen fitter. For more information read the docstring of the optimizer of your choice in `scipy.optimize`.

See also:

- `multifit()`

```python
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 strings 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.

```python
insert(**kwargs)
```

Insert object before index.

```python
load_parameters_from_file(filename)
```

Loads the parameters array from a binary file written with the ‘save_parameters2file’ function.

Parameters **filename** (str) –

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, **kwargs)
Fit the data to the model at all the positions of the navigation dimensions.

Parameters

- **mask** (NumPy array, optional) – To mask (do not fit) at certain position pass a numpy.array of type bool where True indicates that the data will not be fitted at the given position.
- **fetch_only_fixed** (bool) – If True, only the fixed parameters values will be updated when changing the position. Default False.
- **autosave** (bool) – If True, the result of the fit will be saved automatically with a frequency defined by autosave_every. Default False.
- **autosave_every** (int) – Save the result of fitting every given number of spectra. Default 10.
- **show_progressbar** (None or bool) – If True, display a progress bar. If None, the default from the preferences settings is used.
- **interactive_plot** (bool) – 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).
- ****kwargs** (key word arguments) – Any extra key word argument will be passed to the fit method. See the fit method documentation for a list of valid arguments.

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, fancy=True)
Prints the current values of the parameters of all components. :param only_free: If True, only components with free parameters will be printed. Within these, only parameters which are free will be printed.

Parameters

- **only_active** (bool) – If True, only values of active components will be printed
- **component_list** (None or list of components.) – If None, print all components.
• fancy (bool) – If True, attempts to print using html rather than text in the notebook.

**property red_chisq**
Reduced chi-squared. Calculated from self.chisq and self.dof

**remove (thing)**
Remove component from model.

## Examples

```python
definition
>>> s = hs.signals.Signal1D(np.empty(1))
>>> m = s.create_model()
>>> g = hs.model.components1D.Gaussian()
>>> m.append(g)

You could remove g like this

```python
definition
>>> m.remove(g)
```

Like this:

```python
definition
>>> m.remove("Gaussian")
```

Or like this:

```python
definition
>>> 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) –

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 the 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_boundaries ()**
Generate the boundary list.

Necessary before fitting with a boundary aware optimizer.

3.1. hyperspy package
**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 hyperspy components, optional)* – A list of components whose parameters will 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**

```python
>>> 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_mpfit_parameters_info()**

**set_parameters_free** (component_list=None, parameter_name_list=None)

Sets the parameters in a component in a model to free.

**Parameters**
- **component_list** *(None, or list of hyperspy components, 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 strings, 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.

**Examples**

```python
>>> v1 = hs.model.components1D.Voigt()
>>> m.append(v1)
>>> m.set_parameters_free()
>>> m.set_parameters_free(component_list=[v1],
                         parameter_name_list=['area','centre'])
```

See also:
- `set_parameters_not_free()`
- `hyperspy.component.Component.set_parameters_free()`
- `hyperspy.component.Component.set_parameters_not_free()`

**set_parameters_not_free** (component_list=None, parameter_name_list=None)

Sets the parameters in a component in a model to not free.
**Parameters**

- `component_list` *(None, or list of hyperspy components, 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 strings, 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.

**Examples**

```python
g1 = hs.model.components1D.Voigt()
m.append(g1)
m.set_parameters_not_free()

m.set_parameters_not_free(component_list=[g1],
                         parameter_name_list=['area', 'centre'])
```

**See also:**

- `set_parameters_free()`,
- `hyperspy.component.Component`
- `set_parameters_free()`
- `hyperspy.component.Component`
- `set_parameters_not_free()`

**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` *(string)* – Name of the parameter whose value will be changed
- `value` *(number)* – The new value of the parameter
- `component_list` *(list of hyperspy components, optional)* – A list of components whose parameters will 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**

```python
g1 = hs.model.components1D.Voigt()
g2 = hs.model.components1D.Voigt()
m.extend([g1,g2])
m.set_parameters_value('area', 5)
m.set_parameters_value('area', 5, component_list=[g1])
m.set_parameters_value('area', 5, component_list=[g1],
                      only_current=True)
```

**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 parameters array.
    If the parameters array has not being defined yet it creates it filling it with the current parameters.

suspend_update(update_on_resume=True)
    Prevents plot from updating until ‘with’ clause completes.
    See also:
        update_plot()

update_plot(*args, **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.

class hyperspy.model.ModelSpecialSlicers(model, isNavigation)
    Bases: object

hyperspy.roi module

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: hyperspy.roi.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)
    add_widget(signal, axes=None, widget=None, color='green', **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.
**signal** [Signal] The signal to which the widget is added. This is used to determine with plot to add the widget to, and it supplies the axes_manager for the widget.

**axes** [specification of axes to use, default = None] The axes argument specifies which axes the ROI will be applied on. The DataAxis in the collection can be either of the following:

- **a tuple of:**
  - DataAxis. These will not be checked with signal.axes_manager.
  - anything that will index signal.axes_manager

- For any other value, it will check whether the navigation space can fit the right number of axis, and use that if it fits. If not, it will try the signal space.

**widget** [Widget or None (default)] If specified, this is the widget that will be added. If None, the default widget will be used, as given by _get_widget_type().

**color** [Matplotlib color specifier (default: ‘green’)] The color for the widget. Any format that matplotlib uses should be ok. This will not change the color of any widget passed with the ‘widget’ argument.

**kwargs:** All keyword argument are passed to the widget constructor.

**interactive** (signal, navigation_signal='same', out=None, color='green', **kwargs)
Creates an interactively sliced Signal (sliced by this ROI) via hyperspy.interactive.

**signal** [Signal] The source signal to slice

**navigation_signal** [Signal, None or “same” (default)] 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** [Signal] If not None, it will use ‘out’ as the output instead of returning a new Signal.

**color** [Matplotlib color specifier (default: ‘green’)] The color for the widget. Any format that matplotlib uses should be ok. This will not change the color of any widget passed with the ‘widget’ argument.

**kwargs** All kwargs are passed to the roi __call__ method which is called interactively on any roi attribute change.

**remove_widget** (signal)

**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.BasePointROI**
**Bases:** hyperspy.roi.BaseInteractiveROI

Base ROI class for point ROIs, i.e. ROIs with a unit size in each of its dimensions.

**class hyperspy.roi.BaseROI**
**Bases:** traits.has_traits.HasTraits

Base class for all ROIs.

Provides some basic functionality that is 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.

**property ndim**

**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.

```python
class hyperspy.roi.CircleROI(cx, cy, r, r_inner=None)
Bases: hyperspy.roi.BaseInteractiveROI
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 strings 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.

```python
class hyperspy.roi.Line2DROI(x1, y1, x2, y2, linewidth=0)
Bases: hyperspy.roi.BaseInteractiveROI
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

Return type  float

Examples

```python
>>> import hyperspy.api as hs
>>> hs.roi.Line2DROI(0., 0., 1., 2., linewidth=0)
>>> r.angle() 63.4394882292201
```

```python
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 strings 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.

**property length**

**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** (*numeric array, shape (M, N[, C])*) – The image, either grayscale (2D array) or multichannel (3D array, where the final axis contains the channel information).

• **src** (*2-tuple of numeric scalar (float or int)*) – The start point of the scan line.

• **dst** (*2-tuple of numeric scalar (float or int)*) – The end point of the scan line.

• **linewidth** (*int, optional*) – Width of the scan, perpendicular to the line

• **order** (*int in {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** (*string, one of {'constant', 'nearest', 'reflect', 'wrap'}, optional*) – optional How to compute any values falling outside of the image.

• **cval** (*float, optional*) – If mode is ‘constant’, what constant value to use outside the image.

**Returns** return_value – The intensity profile along the scan line. The length of the profile is the ceil of the computed length of the scan line.

**Return type** array

**Examples**

```python
>>> x = np.array([[1, 1, 1, 2, 2, 2]])
>>> img = np.vstack([np.zeros_like(x), x, x, x, np.zeros_like(x)])
>>> img
array([[0, 0, 0, 0, 0, 0],
       [1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2],
       [0, 0, 0, 0, 0, 0]])
>>> profile_line(img, (2, 1), (2, 4))
array([ 1., 1., 2., 2.])
```
Notes

The destination point is included in the profile, in contrast to standard numpy indexing.

class hyperspy.roi.Point1DROI(value)

Bases: hyperspy.roi.BasePointROI

Selects a single point in a 1D space. The coordinate of the point in the 1D space is stored in the ‘value’ trait.

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 strings 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.roi.Point2DROI(x, y)

Bases: hyperspy.roi.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’.

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 strings 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.roi.RectangularROI(left, top, right, bottom)

Bases: hyperspy.roi.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.

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 strings 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.roi.SpanROI (left, right)

Bases: hyperspy.roi.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’.

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 strings 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.

hyperspy.roi.guess_vertical_or_horizontal (axes, signal)

**hyperspy.samfire module**

class hyperspy.samfire.Samfire (model, workers=None, setup=True, **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.

model
The complete model

Type  Model instance

optional_components
A list of components that can be switched off at some pixels if it returns a better Akaike’s Information Criterion with correction (AICc)

Type  list

workers
A number of processes that will perform the fitting parallely

Type  int

pool
A proxy object that manages either multiprocessing or ipyparallel pool

Type  samfire_pool instance

strategies
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.

Type  strategy list

metadata
A dictionary for important samfire parameters

Type  dictionary

active_strategy
The currently active strategy from the strategies list

Type  strategy

update_every
If segmenter strategy is running, updates the historams every time update_every good fits are found.

Type  int

plot_every
When running, samfire plots results every time plot_every good fits are found.

Type  int

save_every
When running, samfire saves results every time save_every good fits are found.

Type  int

start()
start SAMFire
stop()
  stop SAMFire

plot()
  force plot of currently selected active strategy

refresh_database()
  refresh current active strategy database. No previous structure is preserved

backup()
  backs up the current version of the model

change_strategy()
  changes strategy to a new one. Certain rules apply

append()
  appends strategy to the strategies list

extend()
  extends strategies list

remove()
  removes strategy from strategies list

update()
  updates the current model with values, received from a worker

log()
  if _log exists, logs the arguments to the list.

generate_values()
  creates a generator to calculate values to be sent to the workers

property active_strategy
  Returns the active strategy

append(strategy)
  appends the given strategy to the end of the strategies list

Parameters
  strategy (strategy instance) –

backup (filename=None, on_count=True)
  Backs-up the samfire results in a file

Parameters

  • filename (str, None) – the filename. If None, a default value of “backup_”+signal_title is used

  • on_count (bool) – if True (default), 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 | strategy) – index of the new strategy from the strategies list or the strategy object itself

count = 0

extend(iterable)
  extend the strategies list by the given iterable
Parameters iterable (an iterable of strategy instances) –

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”, appends the arguments there

optional_components = []

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) plots 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.

plot_every = 0

pool = None

refresh_database()  
Refreshes currently selected strategy without preserving any “ignored” pixels

remove (thing)  
removes given strategy from the strategies list

Parameters thing (int or strategy instance) – Strategy that is in current strategies list or its index.

running_pixels = []

save_every = nan

start (**kwargs)  
Starts SAMFire.

Parameters **kwargs (key-word arguments) – Any key-word arguments to be passed to Model.fit() call

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, None)) – dictionary of the results. If None, means we are updating in-place (e.g. refreshing the marker or strategies)

• isgood ((bool, 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.samfire.StrategyList

Bases: list

append(thing)
Append object to the end of the list.

extend(iterable)
Extend list by appending elements from the iterable.

remove(thing)
Remove first occurrence of value.
 Raises ValueError if the value is not present.

hyperspy.signal module

class hyperspy.signal.BaseSetMetadataItems

Bases: traits.has_traits.HasTraits

store(*args, **kwargs)

class hyperspy.signal.BaseSignal(data, **kwds)

signal.MVATools

Create a Signal from a numpy array.

Parameters

• data (numpy.ndarray) – The signal data. It can be an array of any dimensions.

• axes (dict, optional) – Dictionary to define the axes (see the documentation of
the AxesManager class for more details).

• attributes (dict, optional) – A dictionary whose items are stored as at-
tributes.

• 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 pa-
rameters that will to stores in the original_metadata attribute. It typically contains
all the parameters that has been imported from the original data file.

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)
Add Gaussian noise to the data.

The operation is performed in-place (i.e. the data of the signal is modified). This method requires a float
data type, otherwise numpy raises a TypeError.

Parameters std(float) – The standard deviation of the Gaussian noise.

Note: This method uses numpy.random.normal() (or dask.array.random.normal() for
lazy signals) to generate the Gaussian noise. In order to seed it, you must use numpy.random.seed().
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** ([`hyperspy.drawing.marker` 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) – If True (default), add the marker to the signal. If False, add the marker to the navigator

- **plot_marker** (bool) – If True (default), plot the marker.

- **permanent** (bool) – If False (default), 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

```python
>>> import scipy.misc
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> m = hs.markers.rectangle(x1=150, y1=100, x2=400, y2=400, color='red')
>>> im.add_marker(m)

Adding to a 1D signal, where the point will change when the navigation index is changed:

```python
>>> s = hs.signals.Signal1D(np.random.random((3, 100)))
>>> marker = hs.markers.point((19, 10, 60), (0.2, 0.5, 0.9))
>>> s.add_marker(marker, permanent=True, plot_marker=True)
```

Add permanent marker:

```python
>>> s = hs.signals.Signal2D(np.random.random((100, 100)))
>>> marker = hs.markers.point(50, 60, color='red')
>>> s.add_marker(marker, permanent=True, plot_marker=True)
```

Add permanent marker to signal with 2 navigation dimensions. The signal has navigation dimensions (3, 2), as the dimensions gets flipped compared to the output from `numpy.random.random()`. To add a vertical line marker which changes for different navigation indices, the list used to make the marker must be a nested list: 2 lists with 3 elements each (2 x 3):

```python
>>> s = hs.signals.Signal1D(np.random.randint(10, size=(3, 100)))
>>> marker = hs.markers.point((10, 30, 50), (30, 50, 60), color='red')
>>> s.add_marker(marker, permanent=True, plot_marker=False)
```

Add permanent marker which changes with navigation position, and do not add it to a current plot:

```python
>>> s = hs.signals.Signal2D(np.random.randint(10, size=(3, 100)))
>>> marker = hs.markers.point((10, 30, 50), (30, 50, 60), color='red')
>>> s.add_marker(marker, permanent=True, plot_marker=False)
>>> s.plot(plot_markers=True) #doctest: +SKIP
```
Removing a permanent marker:

```python
>>> s = hs.signals.Signal2D(np.random.randint(10, size=(100, 100)))
>>> marker = hs.markers.point(10, 60, color='red')
>>> marker.name = "point_marker"
>>> s.add_marker(marker, permanent=True)
>>> del s.metadata.Markers.point Marker
```

Adding many markers as a list:

```python
>>> from numpy.random import random
>>> s = hs.signals.Signal2D(np.random.randint(10, size=(100, 100)))
>>> marker_list = []
>>> for i in range(100):
...     marker = hs.markers.point(random()*100, random()*100, color='red')
...     marker_list.append(marker)
>>> s.add_marker(marker_list, permanent=True)
```

**add_poissonian_noise** (*keep_dtype=True*)

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 a warning is added to the log.

**Parameters**

- **keep_dtype** *(bool)* - 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'. The default is True for convenience.

**Note:** This method uses numpy.random.poisson() (or dask.array.random.poisson() for lazy signals) to generate the Gaussian noise. In order to seed it, you must use numpy.random.seed().

**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** - If inplace=False, returns the apodized signal of the same type as the provided Signal.

**Return type** *BaseSignal* (or subclasses), optional
Examples

```python
>>> import hyperspy.api as hs
>>> holo = hs.datasets.example_signals.object_hologram()
>>> holo.apply_apodization('tukey', tukey_alpha=0.1).plot()
```

**as_lazy**(copy_variance=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

**Returns**
- **res** – The same signal, converted to be lazy

**Return type**
- LazySignal

**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 subclasses) 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:**
- as_signal2D(), transpose(), hyperspy.misc.utils.transpose()

**Examples**

```python
>>> 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 image (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 subclasses) 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:

`as_signal1D()`, `transpose()`, `hyperspy.misc.utils.transpose()`

Examples

```python
gg = s.as_signal2D((0,1))
s.to_signal2D((1,2))
```
```python
>>> s.change_dtype('float')
>>> s.data
array([ 1., 2., 3., 4., 5.])
```

**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.

**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).

**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.

**derivative (axis, order=1, out=None, rechunk=True)**

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 subclasses) 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. If `True` (default), the data may be automatically rechunked before performing this operation.
Returns der – 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.

Return type BaseSignal

See also:
diff(), integrate1D(), integrate_simpson()

diff (axis, order=1, out=None, rechunk=True)

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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns s – 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.

Return type BaseSignal (or subclasses) or None

See also:
derivative(), integrate1D(), integrate_simpson()

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.diff(-1).data.shape
(64,64,1023)
```

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:

\[
\text{Var}[X] = (a \times E[X] + b) \times c
\]
Where \( a \) is the \textit{gain\_factor}, \( b \) is the \textit{gain\_offset} (the Gaussian noise variance) and \( c \) the \textit{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** (\texttt{None} or \texttt{BaseSignal} (or subclasses)) – If \texttt{None}, the signal data is taken as the expected value. Note that this may be inaccurate where the value of \textit{data} is small.

- **gain\_factor** (\texttt{None} or \texttt{float}) – \( a \) in the above equation. Must be positive. If \texttt{None}, take the value from \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model} if defined. Otherwise, suppose pure Poissonian noise (\textit{i.e.} \texttt{gain\_factor=1}). If not \texttt{None}, the value is stored in \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model}.

- **gain\_offset** (\texttt{None} or \texttt{float}) – \( b \) in the above equation. Must be positive. If \texttt{None}, take the value from \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model} if defined. Otherwise, suppose pure Poissonian noise (\textit{i.e.} \texttt{gain\_offset=0}). If not \texttt{None}, the value is stored in \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model}.

- **correlation\_factor** (\texttt{None} or \texttt{float}) – \( c \) in the above equation. Must be positive. If \texttt{None}, take the value from \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model} if defined. Otherwise, suppose pure Poissonian noise (\textit{i.e.} \texttt{correlation\_factor=1}). If not \texttt{None}, the value is stored in \texttt{metadata.Signal.Noise\_properties.Variance\_linear\_model}.

\texttt{fft} (\texttt{shift=False}, \texttt{apodization=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** (\texttt{bool}, \texttt{optional}) – If \texttt{True}, the origin of FFT will be shifted to the centre (default is \texttt{False}).

- **apodization** (\texttt{bool} or \texttt{str}) – Apply an apodization window before calculating the FFT in order to suppress streaks. Valid string values are \{'hann' or 'hamming' or 'tukey'\} If \texttt{True} or \texttt{'hann'}, applies a Hann window. If \texttt{'hamming'} or \texttt{'tukey'}, applies Hamming or Tukey windows, respectively (default is \texttt{False}).

- **\*\*kwargs** (\texttt{dict}) – other keyword arguments are described in \texttt{numpy.fft. fftn()}

Returns  \( s \) – A Signal containing the result of the FFT algorithm

Return type  \textit{ComplexSignal}

Examples

```python
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> 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)

Note: For further information see the documentation of `numpy.fft.fftn()`

fold()
If the signal was previously unfolded, fold it back

get_current_signal(auto_title=True, auto_filename=True)
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.
- **auto_filename** (bool) – If True and `imp_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.

Returns cs – The data at the current coordinates as a Signal

Return type BaseSignal (or subclass)

Examples

```python
>>> im = hs.signals.Signal2D(np.zeros((2, 3, 32, 32)))
>>> im
<Signal2D, title: , dimensions: (3, 2, 32, 32)>  # assuming correct dimensions
>>> im.axes_manager.indices = 2, 1
>>> im.get_current_signal()
<Signal2D, title: (2, 1), dimensions: (32, 32)>  # correct output
```

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_histogram(bins='freedman', range_bins=None, out=None, **kwargs)
Return a histogram of the signal data.

More sophisticated algorithms for determining bins can be used. Aside from the `bins` argument allowing a string specified how bins are computed, the parameters are the same as `numpy.histogram()`.

Parameters

- **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
  - 'scotts': use Scott’s rule to determine bins
  - 'freedman': use the Freedman-diaconis rule to determine bins
  - 'blocks': use bayesian blocks for dynamic bin widths
• `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.

• `out` *(BaseSignal (or subclasses) 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. If True (default), 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` – A 1D spectrum instance containing the histogram.

Return type `Signal1D`

See also: `print_summary_statistics()`, `astroML.density_estimation.histogram()`, `numpy.histogram()`

Notes

• The lazy version of the algorithm does not support the 'knuth' and 'blocks' `bins` arguments.

• The estimators for `bins` are taken from the AstroML project. Read the documentation of `astroML.density_estimation.histogram()` for more info.

Examples

```python
>>> 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()
```

`ifft` *(shift=None, **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).

• `**kwargs` *(dict)* – other keyword arguments are described in `numpy.fft.ifftn()`

Returns `s` – A Signal containing the result of the inverse FFT algorithm

Return type `BaseSignal (or subclasses)`
Examples

```python
>>> import scipy
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> imfft = im.fft()
>>> imfft.ifft()
<Signal2D, title: real(iFFT of FFT of ), dimensions: (512, 512)>
```

Note: For further information see the documentation of `numpy.fft.ifftn()`

`spectrummax (axis, out=None, rechunk=True)`
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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

**Returns** s – A new Signal containing the indices of the maximum along the specified axis.

**Note:** the data dtype is always int.

**Return type** BaseSignal (or subclasses)

See also:
max(), min(), sum(), mean(), std(), var(), indexmin(), valuemax(), valuemin()

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.indexmax(-1).data.shape
(64,64)
```

`indexmin (axis, out=None, rechunk=True)`
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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.
**Returns** `s` – A new Signal containing the indices of the minimum along the specified axis.

Note: the data dtype is always int.

**Return type** `BaseSignal` (or subclasses)

**See also:**

max(), min(), sum(), mean(), std(), var(), indexmax(), valuemax(), valuemin()

**Examples**

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.indexmin(-1).data.shape
(64, 64)
```

`integrate1D(axis, out=None)`

Integrate the signal over the given axis.

The integration is performed using Simpson’s rule if `metadata.Signal.binned` is `False` and simple summation over the given axis if `True`.

**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 subclasses) 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** `s` – A new Signal containing the integral of the provided Signal along the specified axis.

**Return type** `BaseSignal` (or subclasses)

**See also:**

`integrate_simpson()`, `diff()`, `derivative()`

**Examples**

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.integrate1D(-1).data.shape
(64, 64)
```

`integrate_simpson(axis, out=None)`

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 subclasses) 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 `s` – A new *Signal* containing the integral of the provided *Signal* along the specified axis.

Return type *BaseSignal* (or subclasses)

See also:
`diff()`, `derivative()`, `integrate1D()`

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.integrate_simpson(-1).data.shape
(64, 64)
```

**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, parallel=None, inplace=True, ragged=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-py-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.

• **show_progressbar** (*None* or *bool*) – If True, display a progress bar. If None, the default from the preferences settings is used.

• **parallel** (*None* or *bool*) – If True, perform computation in parallel using multiple cores. If None, the default from the preferences settings is used.

• **inplace** (*bool*) – If True (default), the data is replaced by the result. Otherwise a new *Signal* with the results is returned.

• **ragged** (*None* or *bool*) – Indicates if the results for each navigation pixel are of identical shape (and/or numpy arrays to begin with). If None, the appropriate choice is made while processing. Note: None is not allowed for Lazy signals!

• ****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`.

Examples

Apply a Gaussian filter to all the images in the dataset. The sigma parameter is constant:

```python
>>> 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:

```python
>>> 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)
```

`max(axis=None, out=None, rechunk=True)`

Returns a signal with the maximum of the signal along at least one axis.

**Parameters**

- `axis` (`int, str, DataAxis, tuple (of DataAxis) or None`) – 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 subclasses) 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. If `True` (default), the data may be automatically rechunked before performing this operation.

**Returns** `s` – A new Signal containing the maximum of the provided Signal over the specified axes

**Return type** `BaseSignal` (or subclasses)

See also:

- `min()`, `sum()`, `mean()`, `std()`, `var()`, `indexmax()`, `indexmin()`, `valuemax()`, `valuemin()`

**Examples**

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
```
>>> s.max(-1).data.shape
(64,64)

mean \((axis=\text{None}, out=\text{None}, \text{rechunk}=\text{True})\)

Returns a signal with the average of the signal along at least one axis.

Parameters

- **axis** (**int, str, DataAxis, tuple (of DataAxis) or **None****) – 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns **s** – A new Signal containing the mean of the provided Signal over the specified axes

Return type **BaseSignal** (or subclasses)

See also:

- max(), min(), sum(), std(), var(), indexmax(), indexmin(), valuemax(), valuemin()

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.mean(-1).data.shape
(64,64)
```

min \((axis=\text{None}, out=\text{None}, \text{rechunk}=\text{True})\)

Returns a signal with the minimum of the signal along at least one axis.

Parameters

- **axis** (**int, str, DataAxis, tuple (of DataAxis) or **None****) – 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns **s** – A new Signal containing the minimum of the provided Signal over the specified axes

Return type **BaseSignal** (or subclasses)
See also:

max(), sum(), mean(), std(), var(), indexmax(), indexmin(), valuemax(), valuemin()

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.min(-1).data.shape
(64,64)
```

nanmax(\(axis=None, out=None, rechunk=True\))

Identical to max(), except ignores missing (NaN) values. See that method’s documentation for details.

nanmean(\(axis=None, out=None, rechunk=True\))

Identical to mean(), except ignores missing (NaN) values. See that method’s documentation for details.

nanmin(\(axis=None, out=None, rechunk=True\))

Identical to min(), except ignores missing (NaN) values. See that method’s documentation for details.

nanstd(\(axis=None, out=None, rechunk=True\))

Identical to std(), except ignores missing (NaN) values. See that method’s documentation for details.

nansum(\(axis=None, out=None, rechunk=True\))

Identical to sum(), except ignores missing (NaN) values. See that method’s documentation for details.

nanvar(\(axis=None, out=None, rechunk=True\))

Identical to var(), except ignores missing (NaN) values. See that method’s documentation for details.

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.

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.

If None, no navigator will be provided.

Alternatively a BaseSignal (or subclass) instance can be provided. The signal_dimension must be 1 (for a spectrum navigator) or 2 (for a image navigator) and navigation_shape must be 0 (for a static navigator) or 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.

norm [str, optional] The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. (default value is 'auto'). If 'auto', intensity is plotted on a linear scale except when power_spectrum=True (only for complex signals).

**kwargs Only for Signal2D: additional (optional) keyword arguments for matplotlib.pyplot.imshow().

print_summary_statistics (formatter='%.3g', rechunk=True)
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. If True (default), the data may be automatically rechunked before performing this operation.

See also:

get_histogram()

rebin (new_shape=None, scale=None, crop=True, out=None)
Rebin the signal into a smaller or larger shape, based on linear interpolation. Specify either new_shape or scale.

Parameters
- **new_shape (list (of floats or integer) or None)** – For each dimension specify the new_shape. This will internally be converted into a scale parameter.
- **scale** *(list of floats or integer) 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.

  - e.g. 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.

- **out** *(BaseSignal (or subclasses) 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** *s* – The resulting cropped signal.

**Return type** *BaseSignal* (or subclass)

**Examples**

```python
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDXTEM Spectrum, 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)
<EDSTEM Spectrum, title: dimensions (2, 2|2)>
>>> print('Sum = ', sum(sum(sum(test.data))))
Sum = 164.0
```

**rolaxis** *(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
Returns s – Output signal.

Return type BaseSignal (or subclass)

See also:

numpy.roll(), swap_axes()

Examples

```python
>>> 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, **kwds)

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)

`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=None)*

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 `hyperspy.signal.BaseSignal` subclass. Known signal types are signal types that have a specialized `hyperspy.signal.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 run this method without arguments. Note that

**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:**

`print_known_signal_types()`

**Examples**

Let’s first print all known ‘signal_type’s:

```python
>>> 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| hyperspy |
| EDS_SEM           |                     | EDSSEMSpectrum    | hyperspy |
| EDS_TEM           |                     | EDSTEMSpectrum    | hyperspy |
| EELS              | TEM EELS            | EELSSpectrum      | hyperspy |
| hologram          |                     | HologramImage     | hyperspy |
| MySignal          |                     | MySignal          | hspy_ext |
+-------------------+---------------------+-------------------+----------+
```

We can set the `signal_type` using the `signal_type`:
```python
>>> s.set_signal_type("EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
``` or any of its aliases:

```python
>>> s.set_signal_type("TEM EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
```

To set the `signal_type` to `undefined`, simply call the method without arguments:

```python
>>> 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()`, this method will return the former list of signals (information stored in `metadata._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 ints), or int)* – Size of the split parts. If `'auto'`, the `step_sizes` equals one. If an int is given, the splitting is homogeneous.

**Examples**

```python
>>> s = hs.signals.Signal1D(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])
```
HyperSpy Documentation, Release 1.5.1.dev

(continued from previous page)

[<Signal1D, title: , dimensions: (3, 1|2)>,
 <Signal1D, title: , dimensions: (3, 2|2)>]

Returns **split**

- A list of the split signals

**Return type** list

squeeze() Remove single-dimensional entries from the shape of an array and the axes. See `numpy.squeeze()` for more details.

std(axis=None, out=None, rechunk=True) Returns a signal with the standard deviation of the signal along at least one axis.

**Parameters**

- **axis** (int, str, DataAxis, tuple (of DataAxis) or None) – 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

**Returns** s – A new Signal containing the standard deviation of the provided Signal over the specified axes

**Return type** BaseSignal (or subclasses)

See also: `max()`, `min()`, `sum()`, `mean()`, `var()`, `indexmax()`, `indexmin()`, `valuemax()`, `valuemin()`

**Examples**

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.std(-1).data.shape
(64,64)
```

sum(axis=None, out=None, rechunk=True) Sum the data over the given axes.

**Parameters**

- **axis** (int, str, DataAxis, tuple (of DataAxis) or None) – 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 subclasses) 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. If *True* (default), the data may be automatically rechunked before performing this operation.

**Returns** *s* – A new *Signal* containing the sum of the provided Signal along the specified axes.

**Return type** *BaseSignal* (or subclasses)

See also:

`max()`, `min()`, `mean()`, `std()`, `var()`, `indexmax()`, `indexmin()`, `valuemax()`, `valuemin()`

### Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.sum(-1).data.shape
(64, 64)
```

**swap_axes** (*axis1*, *axis2*, *optimize=False*)

Swap two axes in the signal.

**Parameters**

• **axis1** (*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.

• **axis2** (*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.

• **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* – A copy of the object with the axes swapped.

**Return type** *BaseSignal* (or subclass)

See also:

`rollaxis()`

**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.

**Note:** 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.

**See also:**

*T(), as_signal2D(), as_signal1D(), hyperspy.misc.utils.transpose()*

**Examples**

```python
>>> # 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)>

**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** – Whether or not one of the axes needed unfolding (and that unfolding was performed)

Return type **bool**

**Note:** 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** – Whether or not the navigation space needed unfolding (and whether it was performed)

Return type **bool**

**unfold_signal_space()**
Modify the shape of the data to obtain a signal space of dimension 1

Returns **needed_unfolding** – Whether or not the signal space needed unfolding (and whether it was performed)

Return type **bool**

**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**

```python
>>> 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=True*)
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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.
Returns `s` – A new Signal containing the calibrated coordinate values of the maximum along the specified axis.

Return type `BaseSignal` (or subclasses)

See also:

`max()`, `min()`, `sum()`, `mean()`, `std()`, `var()`, `indexmax()`, `indexmin()`, `valuemin()`

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64,64,1024)
>>> s.valuemax(-1).data.shape
(64,64)
```

`valuemin` *(axis, out=None, rechunk=True)*

Returns a signal with the value of coordinates of the minimum along an axis.

Parameters

- **axis** *(int, str, 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns `s` – A new Signal containing the calibrated coordinate values of the minimum along the specified axis.

Return type `BaseSignal` (or subclasses)

See also:

`max()`, `min()`, `sum()`, `mean()`, `std()`, `var()`, `indexmax()`, `indexmin()`, `valuemax()`

`var` *(axis=None, out=None, rechunk=True)*

Returns a signal with the variances of the signal along at least one axis.

Parameters

- **axis** *(int, str, DataAxis, tuple (of DataAxis) or None)* – 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 subclasses) 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. If True (default), the data may be automatically rechunked before performing this operation.

Returns `s` – A new Signal containing the variance of the provided Signal over the specified axes.
Return type: `BaseSignal` (or subclasses)

See also:

- `max()`, `min()`, `sum()`, `mean()`, `std()`, `indexmax()`, `indexmin()`, `valuemax()`, `valuemin()`

Examples

```python
>>> import numpy as np
>>> s = BaseSignal(np.random.random((64,64,1024)))
>>> s.data.shape
(64, 64, 1024)
>>> s.var(-1).data.shape
(64, 64)
```

class hyperspy.signal.MVATools

Bases: `object`

`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 ints)`) – 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

**Note:** The following parameters are only used when **save_figures** = True:

**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 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** parameter is True.

• **save_figures_format** *(str)* – The image format extension.

See also:

*get_bss_factors()* , *get_bss_loadings()*

**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 ints))* – 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

**Note:** The following parameters are only used when **save_figures = True**:

### 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 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** parameter is True.

• **save_figures_format** *(str)* – The image format extension.

See also:

`get_decomposition_factors()`, `get_decomposition_loadings()`

**get_bss_factors()**
Return the blind source separation factors as a `BaseSignal` (or subclass).

See also:

`get_bss_loadings()`, `export_bss_results()`

**get_bss_loadings()**
Return the blind source separation loadings as a `BaseSignal` (or subclass).

See also:

`get_bss_factors()`, `export_bss_results()`

**get_decomposition_factors()**
Return the decomposition factors as a `BaseSignal` (or subclass).

See also:

`get_decomposition_loadings()`, `export_decomposition_results()`

**get_decomposition_loadings()**
Return the decomposition loadings as a `BaseSignal` (or subclass).

See also:

`get_decomposition_factors()`, `export_decomposition_results()`
plot_bss_factors

```python
plot_bss_factors(comp_ids=None, calibrate=True, same_window=True, comp_label=None, per_row=3, title=None)
```

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 ints))` – 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.
- `comp_label (str)` – Will be deprecated in 2.0, please use `title` instead
- `title (str)` – Title of the plot.
- `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

```python
plot_bss_loadings(comp_ids=None, calibrate=True, same_window=True, comp_label=None, with_factors=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', title=None)
```

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 ints))` – 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.
- `comp_label (str)` – Will be deprecated in 2.0, please use `title` instead
- `title (str)` – Title of the plot.
- `with_factors (bool)` – If True, also returns figure(s) with the factors for the given `comp_ids`.
- `cmap (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 (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_decomposition_factors(comp_ids=None, calibrate=True, same_window=True, comp_label=None, cmap=matplotlib.colors.LinearSegmentedColormap object>, per_row=3, title=None)

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 ints)) – 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 plot.
• **cmap (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` parameter is True.

See also:

`plot_decomposition_loadings()`, `plot_decomposition_results()`

```python
plot_decomposition_loadings(comp_ids=None, calibrate=True, same_window=True, comp_label=None, with_factors=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', title=None)
```

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 ints))** – 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 plot.

• **with_factors (bool)** – If True, also returns figure(s) with the factors for the given `comp_ids`.

• **cmap (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 (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()`

```python
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 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_decomposition_factors(), plot_decomposition_loadings(), plot_bss_results()

class hyperspy.signal.ModelManager(signal, dictionary=None)

    Bases: object

    Container for models

class ModelStub(mm, name)

    Bases: object

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()`

```python
>>> 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])
```

### hyperspy.signal_tools module

```python
class hyperspy.signal_tools.BackgroundRemoval(signal, background_type='Power Law', polynomial_order=2, fast=True, plot_remainder=True, zero_fill=False, show_progressbar=None)

Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
```

**apply()**

```python
bg_to_plot (axes_manager=None, fill_with=nan)
```

**create_background_line()**

**create_remainder_line()**

```python
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 strings* 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.

```python
on_disabling_span_selector()
```
rm_to_plot(axes_manager=None, fill_with=nan)

set_background_estimator()

span_selector_changed()

class hyperspy.signal_tools.ButterworthFilter(signal)
Bases: hyperspy.signal_tools.Smoothing

apply()

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 strings 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.

model2plot(axes_manager=None)

class hyperspy.signal_tools.DerivativeTextHandler
Bases: object

legend_artist(legend, orig_handle, fontsize, handlebox)

class hyperspy.signal_tools.DerivativeTextParameters(text, color)
Bases: object

class hyperspy.signal_tools.ImageContrastEditor(image)
Bases: traits.has_traits.HasTraits

apply()

close()

create_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 strings 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.

mpl_help = 'See the matplotlib SymLogNorm for more information.'

plot_histogram()

reset()

span_selector_switch(on)

update_histogram()

update_line()
update_span_selector_traits(*args, **kwargs)

class hyperspy.signal_tools.IntegrateArea(signal, signal_range=None)
    Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
    apply()
    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 strings 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.signal_tools.LineInSignal1D(signal)
    Bases: traits.has_traits.HasTraits
    Adds a vertical draggable line to a spectrum that reports its position to the position attribute of the class.

    position [float] The position of the vertical line in the one dimensional signal. Moving the line changes the
    position but the reverse is not true.

    on [bool] Turns on and off the line

    color [wx.Colour] The color of the line. It automatically redraws the line.

    draw()

    switch_on_off(obj, trait_name, old, new)

    update_position(*args, **kwargs)

class hyperspy.signal_tools.Load
    Bases: traits.has_traits.HasTraits

class hyperspy.signal_tools.Signal1DCalibration(signal)
    Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
    apply()
    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 strings 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.signal_tools.Signal1DRangeSelector(signal)
    Bases: hyperspy.signal_tools.SpanSelectorInSignal1D

class hyperspy.signal_tools.SimpleMessage(text="
    Bases: traits.has_traits.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 strings 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.signal_tools.Smoothing (signal)
Bases: traits.has_traits.HasTraits
close ()
diff_model2plot (axes_manager=None)

property line_color_rgb
plot ()
turn_diff_line_off ()
turn_diff_line_on (diff_order)
update_lines ()
class hyperspy.signal_tools.SmoothingLowess (*args, **kwargs)
Bases: hyperspy.signal_tools.Smoothing
apply ()
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 strings 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.

model2plot (axes_manager=None)
class hyperspy.signal_tools.SmoothingSavitzkyGolay (signal)
Bases: hyperspy.signal_tools.Smoothing
apply ()
diff_model2plot (axes_manager=None)
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 strings 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.

```python
model2plot(axes_manager=None)
```

class hyperspy.signal_tools.SmoothingTV(signal)
    Bases: hyperspy.signal_tools.Smoothing
    apply()
    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 strings 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.

```python
model2plot(axes_manager=None)
```

class hyperspy.signal_tools.SpanSelectorInSignal1D(signal)
    Bases: traits.has_traits.HasTraits
    on_disabling_span_selector()
    reset_span_selector()
    span_selector_switch(on)
    update_span_selector_traits(**args, **kwargs**)

class hyperspy.signal_tools.SpikesRemoval(signal, navigation_mask=None, signal_mask=None)
    Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
    apply()
    create_interpolation_line()
    detect_spike()
    find(back=False)
    get_interpolated_spectrum(axes_manager=None)
    get_interpolation_range()
    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 strings 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.
interpolator = None
on_disabling_span_selector()
span_selector_changed()
update_plot()
update_spectrum_line()

**hyperspy.signals module**

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.

- **EELSSpectrum** For electron energy-loss data with signal_dimension equal 1, i.e. spectral data of n-dimensions. The signal is binned by default.

- **EDSTEMSpectrum** For electron energy-dispersive X-rays data acquired in a transmission electron microscopy with signal_dimension equal 1, i.e. spectral data of n-dimensions. The signal is binned by default.

- **EDSSEMSpectrum** For electron energy-dispersive X-rays data acquired in a scanning electron microscopy with signal_dimension equal 1, i.e. spectral data of n-dimensions. The signal is binned by default.

- **DielectricFunction** For dielectric function data with signal_dimension equal 1. The signal is unbinned by default.

- **HolographyImage** For 2D-images taken via electron holography. Electron wave as ComplexSignal2D can be reconstructed from them.

**hyperspy.ui_registry module**

Registry of user interface widgets.

Format `{“tool_key”: {“toolkit”: <function(obj, display, **kwargs)>}}`

The `tool_key` is defined by the "model function" to which the widget provides and user interface. That function gets the widget function from this registry and executes it passing the `\`\`obj,display and any extra keyword arguments. When `display` is true, `function` displays the widget. If `False` it returns a dictionary with whatever is needed to display the widgets externally (usually for testing or customisation purposes).

`hyperspy.ui_registry.add_gui_method(toolkey)`

3.1. hyperspy package 539
hyperspy.ui_registry.get_gui(self, toolkey, display=True, toolkit=None, **kwargs)

hyperspy.ui_registry.get_partial_gui(toolkey)

### 3.1.3 Module contents

**HyperSpy: a multi-dimensional data analysis package for Python**

Documentation is available in the docstrings and online at [http://hyperspy.org/hyperspy-doc/current/index.html](http://hyperspy.org/hyperspy-doc/current/index.html).

All public packages, functions and classes are in `api`. All other packages and modules are for internal consumption and should not be needed for data analysis.

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:

```python
>>> 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` has been imported as `hs`, `numpy` as `np` and `matplotlib.pyplot` as `plt`.

More details in the `api` docstring.
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.
The website is a fork of the Ipython website.
CITING HYPERSPY

If HyperSpy has been significant to a project that leads to an academic publication, please acknowledge that fact by citing the software. Click on the DOI badge below for formatted citation formats.
CHAPTER
SIX

INDICES AND TABLES

• genindex
• modindex
• search
h
hyperspy, 540
hyperspy._components, 213
hyperspy._components.arctan, 193
hyperspy._components.bleasdale, 194
hyperspy._components.eels_cl_edge, 194
hyperspy._components.eels_double_power_law, 196
hyperspy._components.eels_vignetting, 197
hyperspy._components.error_function, 197
hyperspy._components.exponential, 198
hyperspy._components.expression, 198
hyperspy._components.gaussian, 199
hyperspy._components.gaussian2d, 201
hyperspy._components.gaussianhlf, 201
hyperspy._components.heaviside, 203
hyperspy._components.logistic, 203
hyperspy._components.lorentzian, 204
hyperspy._components.offset, 204
hyperspy._components.pes_core_line_shape, 205
hyperspy._components.pes_see, 205
hyperspy._components.polynomial, 206
hyperspy._components.polynomial_deprecated, 206
hyperspy._components.power_law, 207
hyperspy._components.rc, 208
hyperspy._components.scalable_fixed_pattern, 209
hyperspy._components.skew_normal, 210
hyperspy._components.voigt, 211
hyperspy._components.volume_plasmon_drude, 213
hyperspy._lazy_signals, 459
hyperspy._signals, 269
hyperspy._signals.common_signal1d, 214
hyperspy._signals.common_signal2d, 214
hyperspy._signals.complex_signal, 214
hyperspy._signals.complex_signal1d, 217
hyperspy._signals.complex_signal2d, 218
hyperspy._signals.dielectric_function, 220
hyperspy._signals.edu, 221
hyperspy._signals.edu_sem, 229
hyperspy._signals.edu_tem, 231
hyperspy._signals.eels, 236
hyperspy._signals.hologram_image, 244
hyperspy._signals.lazy, 248
hyperspy._signals.signal1d, 254
hyperspy._signals.signal2d, 264
hyperspy.api, 459
hyperspy.api_nogui, 460
hyperspy.axes, 461
hyperspy.component, 466
hyperspy.components1d, 471
hyperspy.components2d, 472
hyperspy.datasets, 271
hyperspy.datasets.artificial_data, 269
hyperspy.datasets.example_signals, 271
hyperspy.decorators, 472
hyperspy.defaults_parser, 472
hyperspy.docstrings, 272
hyperspy.docstrings.parameters, 272
hyperspy.docstrings.plot, 272
hyperspy.docstrings.signal, 272
hyperspy.docstrings.signal1d, 272
hyperspy.drawing, 298
hyperspy.drawing._markers, 278
hyperspy.drawing._markers.horizontal_line, 272
hyperspy.drawing._markers.horizontal_line_segment, 273
hyperspy.drawing._markers.line_segment, 274
hyperspy.drawing._markers.point, 274
hyperspy.drawing._markers.rectangle, 275
hyperspy.drawing._markers.text, 276
hyperspy.drawing._markers.vertical_line, 277
hyperspy.drawing._markers.vertical_line_segment, 277
hyperspy.drawing._widgets, 283
hyperspy.drawing._widgets.circle, 278
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hyperspy.drawing._widgets.horizontal_line</td>
<td>278</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.label</td>
<td>279</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.line2d</td>
<td>279</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.range</td>
<td>280</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.rectangles</td>
<td>281</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.scalebar</td>
<td>282</td>
</tr>
<tr>
<td>hyperspy.drawing._widgets.vertical_line</td>
<td>283</td>
</tr>
<tr>
<td>hyperspy.drawing.figure</td>
<td>283</td>
</tr>
<tr>
<td>hyperspy.drawing.marker</td>
<td>284</td>
</tr>
<tr>
<td>hyperspy.drawing.mpl_he</td>
<td>285</td>
</tr>
<tr>
<td>hyperspy.drawing.mpl_hie</td>
<td>285</td>
</tr>
<tr>
<td>hyperspy.drawing.mpl_hse</td>
<td>286</td>
</tr>
<tr>
<td>hyperspy.drawing.signal</td>
<td>286</td>
</tr>
<tr>
<td>hyperspy.drawing.signal1d</td>
<td>286</td>
</tr>
<tr>
<td>hyperspy.drawing.tiles</td>
<td>287</td>
</tr>
<tr>
<td>hyperspy.drawing.utils</td>
<td>288</td>
</tr>
<tr>
<td>hyperspy.drawing.widgets</td>
<td>294</td>
</tr>
<tr>
<td>hyperspy.drawing.widgets.widget</td>
<td>298</td>
</tr>
<tr>
<td>hyperspy.events</td>
<td>473</td>
</tr>
<tr>
<td>hyperspy.exceptions</td>
<td>476</td>
</tr>
<tr>
<td>hyperspy.extensions</td>
<td>477</td>
</tr>
<tr>
<td>hyperspy.external</td>
<td>338</td>
</tr>
<tr>
<td>hyperspy.external.astroML</td>
<td>304</td>
</tr>
<tr>
<td>hyperspy.external.astroML.bayesian_block</td>
<td>298</td>
</tr>
<tr>
<td>hyperspy.external.astroML.histtools</td>
<td>300</td>
</tr>
<tr>
<td>hyperspy.external.mpfit</td>
<td>314</td>
</tr>
<tr>
<td>hyperspy.external.mpfit.mpfit</td>
<td>304</td>
</tr>
<tr>
<td>hyperspy.external.mpfit.test_mpfit</td>
<td>313</td>
</tr>
<tr>
<td>hyperspy.external.progressbar</td>
<td>314</td>
</tr>
<tr>
<td>hyperspy.external.tifffile</td>
<td>315</td>
</tr>
<tr>
<td>hyperspy.interactive</td>
<td>477</td>
</tr>
<tr>
<td>hyperspy.io</td>
<td>478</td>
</tr>
<tr>
<td>hyperspy.io_plugins</td>
<td>362</td>
</tr>
<tr>
<td>hyperspy.io_plugins.blockfile</td>
<td>339</td>
</tr>
<tr>
<td>hyperspy.io_plugins.bruker</td>
<td>339</td>
</tr>
<tr>
<td>hyperspy.io_plugins.dens</td>
<td>343</td>
</tr>
<tr>
<td>hyperspy.io_plugins.digital_micrograph</td>
<td>343</td>
</tr>
<tr>
<td>hyperspy.io_plugins.edax</td>
<td>345</td>
</tr>
<tr>
<td>hyperspy.io_plugins.emd</td>
<td>352</td>
</tr>
<tr>
<td>hyperspy.io_plugins.fei</td>
<td>355</td>
</tr>
<tr>
<td>hyperspy.io_plugins.hspy</td>
<td>356</td>
</tr>
<tr>
<td>hyperspy.io_plugins.image</td>
<td>356</td>
</tr>
<tr>
<td>hyperspy.io_plugins.mrc</td>
<td>357</td>
</tr>
<tr>
<td>hyperspy.io_plugins.msa</td>
<td>357</td>
</tr>
<tr>
<td>hyperspy.io_plugins.netcdf</td>
<td>357</td>
</tr>
<tr>
<td>hyperspy.io_plugins.protochips</td>
<td>357</td>
</tr>
<tr>
<td>hyperspy.io_plugins.ripple</td>
<td>358</td>
</tr>
<tr>
<td>hyperspy.io_plugins.sempre_unf</td>
<td>360</td>
</tr>
<tr>
<td>hyperspy.io_plugins.tiff</td>
<td>361</td>
</tr>
<tr>
<td>hyperspy.learn</td>
<td>372</td>
</tr>
<tr>
<td>hyperspy.learn.mlpcan</td>
<td>362</td>
</tr>
<tr>
<td>hyperspy.learn.mva</td>
<td>362</td>
</tr>
<tr>
<td>hyperspy.learn.omf</td>
<td>368</td>
</tr>
<tr>
<td>hyperspy.learn.rpcan</td>
<td>369</td>
</tr>
<tr>
<td>hyperspy.learn.svd_pca</td>
<td>371</td>
</tr>
<tr>
<td>hyperspy.logger</td>
<td>481</td>
</tr>
<tr>
<td>hyperspy.misc</td>
<td>405</td>
</tr>
<tr>
<td>hyperspy.misc.array_tools</td>
<td>386</td>
</tr>
<tr>
<td>hyperspy.misc.config_dir</td>
<td>387</td>
</tr>
<tr>
<td>hyperspy.misc.date_time_tools</td>
<td>387</td>
</tr>
<tr>
<td>hyperspy.misc.ends</td>
<td>376</td>
</tr>
<tr>
<td>hyperspy.misc.ends.ffast_mac</td>
<td>372</td>
</tr>
<tr>
<td>hyperspy.misc.ends.utils</td>
<td>372</td>
</tr>
<tr>
<td>hyperspy.misc.eels</td>
<td>380</td>
</tr>
<tr>
<td>hyperspy.misc.eels.base_gos</td>
<td>376</td>
</tr>
<tr>
<td>hyperspy.misc.eels.eelsdb</td>
<td>376</td>
</tr>
<tr>
<td>hyperspy.misc.eels.eels_effective_angle</td>
<td>377</td>
</tr>
<tr>
<td>hyperspy.misc.eels.hartree_slater_gos</td>
<td>378</td>
</tr>
<tr>
<td>hyperspy.misc.eels.hydrogenic_gos</td>
<td>379</td>
</tr>
<tr>
<td>hyperspy.misc.eels.tools</td>
<td>379</td>
</tr>
<tr>
<td>hyperspy.misc.elements</td>
<td>389</td>
</tr>
<tr>
<td>hyperspy.misc.example_signals_loading</td>
<td>389</td>
</tr>
<tr>
<td>hyperspy.misc.export_dictionary</td>
<td>390</td>
</tr>
<tr>
<td>hyperspy.misc.holograph</td>
<td>383</td>
</tr>
<tr>
<td>hyperspy.misc.holograph.reconstruct</td>
<td>381</td>
</tr>
<tr>
<td>hyperspy.misc.holograph.tools</td>
<td>382</td>
</tr>
<tr>
<td>hyperspy.misc.io</td>
<td>385</td>
</tr>
<tr>
<td>hyperspy.misc.io.fei_stream_readers</td>
<td>383</td>
</tr>
<tr>
<td>hyperspy.misc.io.tools</td>
<td>384</td>
</tr>
<tr>
<td>hyperspy.misc.io.utils_readfile</td>
<td>384</td>
</tr>
<tr>
<td>hyperspy.misc.ipython_tools</td>
<td>391</td>
</tr>
<tr>
<td>hyperspy.misc.machine_learning</td>
<td>385</td>
</tr>
<tr>
<td>hyperspy.misc.machine_learning.import_sklearn</td>
<td>385</td>
</tr>
<tr>
<td>hyperspy.misc.machine_learning.orthomax</td>
<td>385</td>
</tr>
<tr>
<td>hyperspy.misc.machine_learning.tools</td>
<td>385</td>
</tr>
<tr>
<td>hyperspy.misc.material</td>
<td>391</td>
</tr>
<tr>
<td>hyperspy.misc.math_tools</td>
<td>394</td>
</tr>
<tr>
<td>hyperspy.misc.model_tools</td>
<td>394</td>
</tr>
<tr>
<td>hyperspy.misc.physical_constants</td>
<td>395</td>
</tr>
<tr>
<td>hyperspy.misc.physics_tools</td>
<td>395</td>
</tr>
<tr>
<td>hyperspy.misc.rgb_tools</td>
<td>395</td>
</tr>
<tr>
<td>hyperspy.misc.signal_tools</td>
<td>396</td>
</tr>
<tr>
<td>hyperspy.misc.slicing</td>
<td>396</td>
</tr>
<tr>
<td>hyperspy.misc.test_utils</td>
<td>397</td>
</tr>
<tr>
<td>hyperspy.misc.tv_denoise</td>
<td>398</td>
</tr>
<tr>
<td>hyperspy.misc.utils</td>
<td>399</td>
</tr>
</tbody>
</table>
hyperspy.model, 482
hyperspy.models, 423
hyperspy.models.edsmode, 406
hyperspy.models.edsemmode, 410
hyperspy.models.edstemmode, 410
hyperspy.models.eelsmode, 411
hyperspy.models.model1d, 416
hyperspy.models.model2d, 422
hyperspy.Release, 459
hyperspy.roi, 492
hyperspy.samfire, 497
hyperspy.samfire_utils, 430
hyperspy.samfire_utils.fit_tests, 425
hyperspy.samfire_utils.global_strategies, 425
hyperspy.samfire_utils.goodness_of_fit_tests, 425
hyperspy.samfire_utils.global_strategies, 425
hyperspy.samfire_utils.goodness_of_fit_tests, 425
hyperspy.samfire_utils.segmenters, 424
hyperspy.samfire_utils.segmenters.histogram, 424
hyperspy.samfire_utils.strategy, 428
hyperspy.samfire_utils.weights, 425
hyperspy.samfire_utils.weights.red_chisq, 424
hyperspy.signal, 501
hyperspy.signal_tools, 534
hyperspy.signals, 539
hyperspy.tests, 455
hyperspy.tests.axes, 431
hyperspy.tests.axes.test_axes_manager, 431
hyperspy.tests.component, 435
hyperspy.tests.component.test_bleasdale, 432
hyperspy.tests.component.test_component, 432
hyperspy.tests.component.test_component ActiveArray, 432
hyperspy.tests.component.test_component_set_parameters, 433
hyperspy.tests.component.test_components2D, 433
hyperspy.tests.component.test_double_power_law, 434
hyperspy.tests.component.test_exponential, 434
hyperspy.tests.component.test_gaussian2d, 434
hyperspy.tests.component.test_logistic, 434
hyperspy.tests.component.test_lorentzian, 434
hyperspy.tests.component.test_rc, 434
hyperspy.tests.component.test_volume_plasmon_drude, 435
hyperspy.tests.datasets, 435
hyperspy.tests.datasets.test_artificial_data, 435
hyperspy.tests.io, 438
hyperspy.tests.io.generate_dm_testing_files, 436
hyperspy.tests.io.test_dm_stackbuilder_plugin, 436
hyperspy.tests.misc, 439
hyperspy.tests.misc.test_date_time_tools, 438
hyperspy.tests.misc.test_math_tools, 438
hyperspy.tests.misc.test_rgbtools, 438
hyperspy.tests.misc.test_utils, 439
hyperspy.tests.mva, 443
hyperspy.tests.mva.test_export, 442
hyperspy.tests.mva.test_onmf, 442
hyperspy.tests.samfire, 445
hyperspy.tests.samfire.test_goodness_of_fit_tests, 443
hyperspy.tests.samfire.test_histogram_segmenter, 443
hyperspy.tests.samfire.test_red_chisq_weight, 443
hyperspy.tests.samfire.test_strategy, 444

Python Module Index 551
hyperspy.tests.samfire.test_strategy_list, 444
hyperspy.tests.samfire.test_utils, 445
hyperspy.tests.signal, 453
hyperspy.tests.signal.test_assign_subclass, 445
hyperspy.tests.signal.test_binned, 446
hyperspy.tests.signal.test_complex_signal2d, 447
hyperspy.tests.signal.test_eds_sem, 447
hyperspy.tests.signal.test_eds_tem, 448
hyperspy.tests.signal.test_find_peaks1D_ohaver, 450
hyperspy.tests.signal.test_folding, 451
hyperspy.tests.signal.test_image, 452
hyperspy.tests.signal.test_spectrum, 453
hyperspy.tests.test_dictionary_tree_browser, 455
hyperspy.tests.utils, 455
hyperspy.tests.utils.test_eds, 454
hyperspy.tests.utils.test_material, 454
hyperspy.tests.utils.test_print_known_signal_types, 454
hyperspy.tests.utils.test_slugify, 454
hyperspy.ui_registry, 539
hyperspy.utils, 458
hyperspy.utils.eds, 455
hyperspy.utils.markers, 455
hyperspy.utils.material, 455
hyperspy.utils.model, 456
hyperspy.utils.model_selection, 456
hyperspy.utils.parallel_pool, 456
hyperspy.utils.plot, 458
hyperspy.utils.roi, 458
hyperspy.utils.samfire, 458
INDEX

A

A (hyperspy_components.pes_see.SEE attribute), 205
A() (hyperspy_components.gaussianhf.GaussianHF property), 202
active_is_multidimensional() (hyperspy.component.Component property), 466
active_strategy (hyperspy.samfire.Samfire attribute), 498
active_strategy() (hyperspy.samfire.Samfire property), 499
add() (hyperspy.events.EventSuppressor method), 475
add_data() (hyperspy.drawing.marker.MarkerBase method), 284
add_dictionary() (hyperspy.misc.utils.DictionaryTreeBrowser method), 400
add_elements() (hyperspy_signals.eds.EDS_mixin method), 221
add_elements() (hyperspy_signals.eels.EELSSpectrum_mixin method), 236
add_family_lines() (hyperspy.models.edmodel.EDSMExcitement method), 406
add_filename_to_general() (hyperspy.io_plugins.bruker.BCF_reader method), 339
add_gaussian_noise() (hyperspy.signal.BaseSignal method), 501
add_gui_method() (in module hyperspy.ui_registry), 539
add_jobs() (hyperspy.samfire_utils.samfire_pool.SamfirePool method), 426, 427
add_line() (hyperspy.drawing.signal1d.Signal1DFigure method), 286
add_lines() (hyperspy_signals.eds.EDS_mixin method), 221
add_marker() (hyperspy.drawing.figure.BlittedFigure method), 283
add_marker() (hyperspy.signal.BaseSignal method), 501
add_node() (hyperspy.misc.utils.DictionaryTreeBrowser method), 400
add_phase_ramp() (hyperspy_signals.complex_signal2d.Complex2Dmixin method), 218
add_poissonian_noise() (hyperspy.signal.BaseSignal method), 503
add_polynomial_background() (hyperspy_models.edmodel.EDSMExcitement method), 406
add_ramp() (hyperspy_signals.signal2d.Signal2D method), 264
add_right_pointer() (hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer method), 286
add_scalar_axis() (in module hyperspy.misc.utils), 401
add_signal() (hyperspy.io_plugins.emd.EMD method), 353
add_signal_range() (hyperspy_models.model1d.Model1D method), 419
add_signal_range() (hyperspy_models.model2d.Model2D method), 423
add_widget() (hyperspy.roi.BaseInteractiveROI method), 492
add_xray_lines_markers() (hyperspy_signals.eds.EDS_mixin method), 222
AIC() (in module hyperspy.ui_registry.model_selection), 456
AIC_test (class in hyperspy.samfire_utils.goodness_of_fit_tests.information_theory), 423
AICC() (in module hyperspy.ui_registry.model_selection), 456
AICC_test (class in hyperspy.samfire_utils.goodness_of_fit_tests.information_theory), 423
align1D() (hyperspy_signals.signal1d.Signal1D method), 254
align2D() (hyperspy_signals.signal2d.Signal2D method), 264
align_zero_loss_peak() (hyperspy_signals.eels.EDLSSpectrum_mixin method), 222
asarray() (hyperspy.external.tifffile.TiffFrame method), 335
asarray() (hyperspy.external.tifffile.TiffPage method), 332
asarray() (hyperspy.external.tifffile.TiffSequence method), 329
aspage() (hyperspy.external.tifffile.TiffFrame method), 335
aspage() (hyperspy.external.tifffile.TiffPage method), 332
asrgb() (hyperspy.external.tifffile.TiffFrame method), 335
asrgb() (hyperspy.external.tifffile.TiffPage method), 332
assert_deep_almost_equal() (in module hyperspy.misc.test_utils), 397
assertwarn() (in module hyperspy.misc.test_utils), 397
assign_current_value_to_all() (hyperspy.component.Parameter method), 470
assign_current_values_to_all() (hyperspy.model.BaseModel method), 485
assign_pointer() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 285
assign_signal_subclass() (in module hyperspy.io), 478
atomic_to_weight() (in module hyperspy.misc.material), 391
attrsetter() (in module hyperspy.misc.utils), 401
auto_update (hyperspy.drawing.signal1D.Signal1DLine attribute), 287
auto_update_plot() (hyperspy.drawing.mpl_bse.MPL_HyperSignal1D_Explorer property), 286
axes (hyperspy.external.tifffile.TiffPage attribute), 331
axes (hyperspy.external.tifffile.TiffSequence attribute), 329
axes() (hyperspy.drawing.widget.WidgetBase property), 297
axes_are_aligned_with_data() (hyperspy.axes.AxesManager property), 462
axes_ticks() (hyperspy.drawing.image.ImagePlot property), 283
AxesManager (class in hyperspy.axes), 461

B (hyperspy_components.pes_see.SEE attribute), 205
BackgroundRemoval (class in hyperspy.signal_tools), 534
backup() (hyperspy.samfire.Samfire method), 499
BaseInteractiveROI (class in hyperspy.roi), 492
BaseModel (class in hyperspy.model), 482
BasePointROI (class in hyperspy.roi), 493
BaseROI (class in hyperspy.roi), 493
BaseSetMetadataItems (class in hyperspy.signal), 501
BaseSignal (class in hyperspy.signal), 501
bayesian_blocks() (in module hyperspy.external.astroML.bayesian_blocks), 299
bcf_hyperspectra() (in module hyperspy.io_plugins.bruker), 341
bcf_images() (in module hyperspy.io_plugins.bruker), 342
BCF_reader (class in hyperspy.io_plugins.bruker), 339
bcef_reader() (in module hyperspy.io_plugins.bruker), 342
bg_to_plot() (hyperspy.signal_tools.BackgroundRemoval method), 534
BIC() (in module hyperspy.utils.model_selection), 456
BIC_test (class in hyperspy.samfire_utils.goodness_of_fit_tests.information_theory), 423
bins() (hyperspy.extenal.astroML.histtools.KnuthF method), 301
bitspersample (hyperspy.external.tifffile.TiffPage attribute), 332
blas_enorm32 (hyperspy.external.mpfit.mpfit.mpfit attribute), 312
blas_enorm64 (hyperspy.external.mpfit.mpfit.mpfit attribute), 312
Bleasdale (class in hyperspy._components.bleasdale), 194
blind_source_separation() (hyperspy.learn.mva.MVA method), 363
BlittedFigure (class in hyperspy.drawing.figure), 283
bragg_scattering_angle() (in module hyperspy.misc.physics_tools), 395
broadcast_signals() (in module hyperspy.misc.signal_tools), 396
bss_algorithm(hyperspy.learn.mva.LearningResults attribute), 362
bss_factors (hyperspy.learn.mva.LearningResults attribute), 362
bss_loadings (hyperspy.learn.mva.LearningResults attribute), 363
ButterworthFilter (class in hyperspy.signal_tools), 535
button_release() (hyperspy.drawing.widget.DraggableWidgetBase method), 295
button_release() (hyperspy.drawing.widget.ResizableDraggableWidgetBase method), 295

Index  555
byteorder() (hyperspy.external.tifffile.TiffFile property), 324
ByteOrderError, 476

C
calc_covar() (hyperspy.external.mpfit.mpfit.mpfit method), 313
calc_real_time() (hyperspy.io_plugins.bruker.HyperHeader method), 340
calculate_bins_histogram() (in module hyperspy.misc.array_tools), 386
calculate_carrier_frequency() (in module hyperspy.misc.holography.tools), 382
calculate_chunks() (in module hyperspy.io_plugins.emd), 354
calculate_line_position() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 282
calculate_size() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 282
calculate_text_position() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 282
calibrate() (hyperspy._signals.signal1d.Signal1D method), 255
calibrate() (hyperspy.axes.DataAxis method), 464
calibrate_energy_axis() (hyperspy.models.edsmode.EDSModel method), 407
calculate_xray_lines() (hyperspy.models.edsmode.EDSModel method), 407
call() (hyperspy.external.mpfit.mpfit.mpfit method), 313
centering_and_whitening() (in module hyperspy.learn.mva), 368
centre (hyperspy._components.voigt.Voigt attribute), 211
centre (hyperspy.learn.mva.LearningResults attribute), 363
centre_colormap_values() (in module hyperspy.drawing.utils), 288
change_dtype() (hyperspy.signals.complex_signal.ComplexSignal mixin method), 215
change_dtype() (hyperspy.signals.lazy.LazySignal method), 248
change_dtype() (hyperspy.signal.BaseSignal method), 505
change_strategy() (hyperspy.samfire.Samfire method), 499
change_timestep() (hyperspy.samfire_utils.samfire_worker.Worker method), 428
check_data_tag_delimiter() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 343
check_index_valid() (hyperspy.io_plugins.bruker.BCF_reader method), 339
check_running_tests_in_CI() (in module hyperspy.misc.test_utils), 398
check_that_flags_make_sense() (in module hyperspy.misc.export_dictionary), 390
chisq (hyperspy.model.BaseModel attribute), 482
chisq (hyperspy.models.model1d.Model1D attribute), 417
chisq (hyperspy.models.model2d.Model2D attribute), 422
CircleROI (class in hyperspy.roi), 494
CircleWidget (class in hyperspy.drawing._widgets.circle), 278
clean() (hyperspy.samfire_utils.strategy.GlobalStrategy method), 428
clean() (hyperspy.samfire_utils.strategy.LocalStrategy method), 429
close() (hyperspy.drawing.figure.BlittedFigure method), 283
close() (hyperspy.drawing.marker.MarkerBase method), 284
close() (hyperspy.drawing.mpl.he.MPL_HyperExplorer method), 285
close() (hyperspy.drawing.signal1d.Signal1DLine method), 287
close() (hyperspy.drawing.tiles.HistogramTilePlot method), 287
close() (hyperspy.drawing.widget.WidgetBase method), 297
close() (hyperspy.external.tifffile.FileHandle method), 330
close() (hyperspy.external.tifffile.TiffFile method), 324
close() (hyperspy.external.tifffile.TiffSequence method), 329
close() (hyperspy.signal_tools.ImageContrastEditor method), 535
close() (hyperspy.signal_tools.Smoothing method), 537
close_file() (hyperspy._signals.lazy.LazySignal method), 249
close_navigator_plot() (hyperspy.drawing.mpl.he.MPL_HyperExplorer method), 285
close_plot (hyperspy.samfire_utils.strategy.SamfireStrategy method), 420
Index
<table>
<thead>
<tr>
<th>Function</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinates()</td>
<td>hyperspy.axes.AxesManager property</td>
<td>463</td>
</tr>
<tr>
<td>copy()</td>
<td>hyperspy.axes.AxesManager method</td>
<td>463</td>
</tr>
<tr>
<td>copy()</td>
<td>hyperspy.axes.DataAxis method</td>
<td>464</td>
</tr>
<tr>
<td>copy()</td>
<td>hyperspy.misc.utils.DictionaryTreeBrowser method</td>
<td>400</td>
</tr>
<tr>
<td>copy()</td>
<td>hyperspy.signal.BaseSignal method</td>
<td>506</td>
</tr>
<tr>
<td>copy_slice_from_whitelist()</td>
<td>in module hyperspy.misc.slicing</td>
<td>396</td>
</tr>
<tr>
<td>correct_INCA_format()</td>
<td>in module hyperspy.io_plugins.ripple</td>
<td>358</td>
</tr>
<tr>
<td>count</td>
<td>hyperspy.external.tifffile.TiffTag attribute</td>
<td>336</td>
</tr>
<tr>
<td>count</td>
<td>hyperspy.samfire.Samfire attribute</td>
<td>499</td>
</tr>
<tr>
<td>create_artificial_samfire()</td>
<td>in module hyperspy.tests.samfire.test_strategy</td>
<td>444</td>
</tr>
<tr>
<td>create_axes()</td>
<td>hyperspy.axes.AxesManager method</td>
<td>463</td>
</tr>
<tr>
<td>create_axis()</td>
<td>hyperspy.drawing.image.ImagePlot method</td>
<td>283</td>
</tr>
<tr>
<td>create_axis()</td>
<td>hyperspy.drawing.tiles.HistogramTilePlot method</td>
<td>287</td>
</tr>
<tr>
<td>create_axis()</td>
<td>hyperspy.signal_tools.ImageContrastEditor method</td>
<td>535</td>
</tr>
<tr>
<td>create_background_line()</td>
<td>hyperspy.signal_tools.BackgroundRemoval method</td>
<td>534</td>
</tr>
<tr>
<td>create_figure()</td>
<td>hyperspy.drawing.figure.BlittedFigure method</td>
<td>283</td>
</tr>
<tr>
<td>create_figure()</td>
<td>hyperspy.drawing.image.ImagePlot method</td>
<td>283</td>
</tr>
<tr>
<td>create_figure()</td>
<td>in module hyperspy.drawing.utils</td>
<td>288</td>
</tr>
<tr>
<td>create_interpolation_line()</td>
<td>hyperspy.signal_tools.SpikesRemoval method</td>
<td>538</td>
</tr>
<tr>
<td>create_map_objects()</td>
<td>in module hyperspy.misc.utils</td>
<td>402</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.signals.eds_sem.EDSSEM_mixin method</td>
<td>230</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.signals.eds_tem.EDSTEM_mixin method</td>
<td>232</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.signals.eels.EELSSpectrum_mixin method</td>
<td>237</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.signals.signal1d.Signal1D method</td>
<td>255</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.signals.signal2d.Signal2D method</td>
<td>265</td>
</tr>
<tr>
<td>create_model()</td>
<td>hyperspy.samfire_utils.samfire_worker.Worker method</td>
<td>428</td>
</tr>
<tr>
<td>create_output()</td>
<td>in module hyperspy.external.tifffile</td>
<td>337</td>
</tr>
<tr>
<td>create_remainder_line()</td>
<td>hyperspy.signal_tools.BackgroundRemoval method</td>
<td>534</td>
</tr>
<tr>
<td>create_right_axis()</td>
<td>hyperspy.drawing.signal1d.Signal1DFigure method</td>
<td>286</td>
</tr>
<tr>
<td>create_samfire()</td>
<td>hyperspy.model.BaseModel method</td>
<td>485</td>
</tr>
<tr>
<td>create_worker()</td>
<td>in module hyperspy.samfire_utils.samfire_worker</td>
<td>428</td>
</tr>
<tr>
<td>crop()</td>
<td>hyperspy.signal.BaseSignal method</td>
<td>506</td>
</tr>
<tr>
<td>crop_decomposition_dimension()</td>
<td>hyperspy.learn.mva.LearningResults method</td>
<td>363</td>
</tr>
<tr>
<td>crop_image()</td>
<td>hyperspy.signals.signal2d.Signal2D method</td>
<td>265</td>
</tr>
<tr>
<td>crop_signal1D()</td>
<td>hyperspy.signals.signal1d.Signal1D method</td>
<td>255</td>
</tr>
<tr>
<td>current_component_values</td>
<td>class in hyperspy.misc.model_tools</td>
<td>394</td>
</tr>
<tr>
<td>current_model_values</td>
<td>class in hyperspy.misc.model_tools</td>
<td>395</td>
</tr>
</tbody>
</table>

**D**

dasky_freedman_bin_width() | in module hyperspy.external.astroML.histtools | 301 |
dasky_histogram() | in module hyperspy.external.astroML.histtools | 301 |
dasky_scotts_bin_width() | in module hyperspy.external.astroML.histtools | 302 |
data | hyperspy.io_plugins.semper_unf.SemperFormat attribute | 360 |
data() | hyperspy.signal.BaseSignal property | 506 |
DataAxis | class in hyperspy.axes | 464 |
datatime_to_serial_date() | hyperspy.misc. date_time tools | 388 |
decomposition() | hyperspy.signals.eds_tem.EDSTEM_mixin method | 232 |

558 Index
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>decomposition()</td>
<td>hyperspy.signals.lazy.LazySignal method</td>
<td>249</td>
</tr>
<tr>
<td>decomposition()</td>
<td>hyperspy.learn.mva.MVA method</td>
<td>364</td>
</tr>
<tr>
<td>decomposition_algorithm</td>
<td>hyperspy.learn.mva.LearningResults attribute</td>
<td>363</td>
</tr>
<tr>
<td>decrease_size()</td>
<td>hyperspy.drawing.widgets.circle.CircleWidget method</td>
<td>278</td>
</tr>
<tr>
<td>decrease_size()</td>
<td>hyperspy.drawing.widgetResizableDraggableWidgetBase method</td>
<td>295</td>
</tr>
<tr>
<td>deepcopy()</td>
<td>hyperspy.axes.AxesManager method</td>
<td>463</td>
</tr>
<tr>
<td>deepcopy()</td>
<td>hyperspy.misc.utils.DictionaryTreeBrowser method</td>
<td>400</td>
</tr>
<tr>
<td>deepcopy()</td>
<td>hyperspy.signal.BaseSignal method</td>
<td>506</td>
</tr>
<tr>
<td>default_traits_view()</td>
<td>hyperspy.component.Parameter method</td>
<td>470</td>
</tr>
<tr>
<td>default_value</td>
<td>hyperspy.component.NoneFloat attribute</td>
<td>468</td>
</tr>
<tr>
<td>derivative()</td>
<td>hyperspy.signal.BaseSignal method</td>
<td>506</td>
</tr>
<tr>
<td>DenseSliceCOO</td>
<td>hyperspy.io.plugins.fei_stream_readers</td>
<td>383</td>
</tr>
<tr>
<td>density_of_mixture()</td>
<td>hyperspy.misc.material</td>
<td>391</td>
</tr>
<tr>
<td>deprecation_warning()</td>
<td>hyperspy.io_plugins.emd.FeiEMDReader attribute</td>
<td>354</td>
</tr>
<tr>
<td>detect_spike()</td>
<td>hyperspy.signal_tools.SpikesRemoval method</td>
<td>333</td>
</tr>
<tr>
<td>detect_spike()</td>
<td>hyperspy.signal_tools.EELSModel method</td>
<td>333</td>
</tr>
<tr>
<td>dict2dgroup()</td>
<td>hyperspy.io_plugins.hspy</td>
<td>356</td>
</tr>
<tr>
<td>dict2marker()</td>
<td>hyperspy.drawing.marker</td>
<td>285</td>
</tr>
<tr>
<td>dict2array()</td>
<td>hyperspy.misc.array_tools</td>
<td>386</td>
</tr>
<tr>
<td>dict2signal()</td>
<td>hyperspy.io</td>
<td>478</td>
</tr>
<tr>
<td>dictionaries</td>
<td>hyperspy.io_plugins.emd.FeiEMDReader attribute</td>
<td>354</td>
</tr>
<tr>
<td>dictionarize()</td>
<td>hyperspy.io_plugins.brucker</td>
<td>342</td>
</tr>
<tr>
<td>dictionary_from_template()</td>
<td>hyperspy.defaults_parser</td>
<td>473</td>
</tr>
<tr>
<td>DictionaryTreeBrowser</td>
<td>hyperspy.misc.utils</td>
<td>399</td>
</tr>
<tr>
<td>DielectricFunction</td>
<td>hyperspy.signals.dielectric_function</td>
<td>220</td>
</tr>
<tr>
<td>DielectricFunctionMixin</td>
<td>hyperspy.signals.dielectric_function</td>
<td>220</td>
</tr>
<tr>
<td>diff()</td>
<td>hyperspy.signals.lazy.LazySignal method, hyperspy.signal.BaseSignal method</td>
<td>507</td>
</tr>
<tr>
<td>diff_model2plot()</td>
<td>hyperspy.signal_tools.Smoothing method</td>
<td>537</td>
</tr>
<tr>
<td>DigitalMicrographReader</td>
<td>hyperspy.io_plugins.digital_micrograph</td>
<td>343</td>
</tr>
<tr>
<td>dimension_array_dtype()</td>
<td>hyperspy.io_plugins.fei</td>
<td>355</td>
</tr>
<tr>
<td>dirname()</td>
<td>hyperspy.external.tifffile.FileHandle property</td>
<td>330</td>
</tr>
<tr>
<td>disable_adjust_position()</td>
<td>hyperspy.models.model1d.Model1D method</td>
<td>419</td>
</tr>
<tr>
<td>disable_adjust_position()</td>
<td>hyperspy.models.model2d.Model2D method</td>
<td>423</td>
</tr>
<tr>
<td>disable_background()</td>
<td>hyperspy.models.eelsmodel.EELSMETHOD</td>
<td>411</td>
</tr>
<tr>
<td>disable_edges()</td>
<td>hyperspy.models.eelsmodel.EELSMETHOD</td>
<td>411</td>
</tr>
<tr>
<td>disable_fine_structure()</td>
<td>hyperspy.models.eelsmodel.EELSMETHOD</td>
<td>412</td>
</tr>
<tr>
<td>disable_free_onset_energy()</td>
<td>hyperspy.models.eelsmodel.EELSMETHOD</td>
<td>412</td>
</tr>
<tr>
<td>disable_plot_components()</td>
<td>hyperspy.models.BaseModel method</td>
<td>485</td>
</tr>
<tr>
<td>disable_plot_components()</td>
<td>hyperspy.models.model1d.Model1D method</td>
<td>419</td>
</tr>
<tr>
<td>disable_xray_lines()</td>
<td>hyperspy.models.edsmmodel.EDSMModel method</td>
<td>407</td>
</tr>
<tr>
<td>disconnect()</td>
<td>hyperspy.drawing.widgets.range.RangeWidget method</td>
<td>281</td>
</tr>
<tr>
<td>disconnect()</td>
<td>hyperspy.drawing.image.ImagePlot method</td>
<td>284</td>
</tr>
<tr>
<td>Method/Function</td>
<td>Class/Module</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>disconnect()</td>
<td>hyperspy.drawing.widget.WidgetBase method</td>
<td>297</td>
</tr>
<tr>
<td>disconnect()</td>
<td>hyperspy.events.Event method</td>
<td>474</td>
</tr>
<tr>
<td>disconnect()</td>
<td>hyperspy.drawing.widget.WidgetBase method</td>
<td>297</td>
</tr>
<tr>
<td>DM3DataTypeError</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>DM3FileVersionError</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>DM3TagError</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>DM3TagIDError</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>DM3TagTypeError</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>dof()</td>
<td>hyperspy.models.BaseModel attribute</td>
<td>482</td>
</tr>
<tr>
<td>dof()</td>
<td>hyperspy.models.model1d.Model1D attribute</td>
<td>417</td>
</tr>
<tr>
<td>dof()</td>
<td>hyperspy.models.model2d.Model2D attribute</td>
<td>422</td>
</tr>
<tr>
<td>DoublePowerLaw</td>
<td>class in hyperspy_components.eels_double_power_law</td>
<td>196</td>
</tr>
<tr>
<td>DraggableWidgetBase</td>
<td>class in hyperspy.drawing.widget</td>
<td>294</td>
</tr>
<tr>
<td>draw()</td>
<td>hyperspy.signal_tools.LineInSignal1D method</td>
<td>536</td>
</tr>
<tr>
<td>draw_patch()</td>
<td>hyperspy.drawing.widgets.range.ModifiableSpanSelector method</td>
<td>280</td>
</tr>
<tr>
<td>draw_patch()</td>
<td>hyperspy.drawing.widget.WidgetBase method</td>
<td>297</td>
</tr>
<tr>
<td>dtype()</td>
<td>hyperspy.external.tifffile.TiffPage attribute</td>
<td>431</td>
</tr>
<tr>
<td>dtype()</td>
<td>hyperspy.external.tifffile.TiffTag attribute</td>
<td>336</td>
</tr>
<tr>
<td>dtype()</td>
<td>hyperspy.io_plugins.digital_micrograph.ImageObject property</td>
<td>336</td>
</tr>
<tr>
<td>dtype()</td>
<td>hyperspy.io_plugins.digital_micrograph.ImageObject property</td>
<td>344</td>
</tr>
<tr>
<td>dummy()</td>
<td>hyperspy.drawing.widgets.range.ModifiableSpanSelector</td>
<td>280</td>
</tr>
<tr>
<td>dummy()</td>
<td>hyperspy.io_plugins.digital_micrograph.ImageObject property</td>
<td>466</td>
</tr>
<tr>
<td>dummy()</td>
<td>hyperspy.io_plugins.digital_micrograph.ImageObject property</td>
<td>466</td>
</tr>
<tr>
<td>dump_dictionary()</td>
<td>hyperspy.misc.io.tools</td>
<td>384</td>
</tr>
<tr>
<td>EDSTEM_mixin</td>
<td>class in hyperspy_signals.eds</td>
<td>232</td>
</tr>
<tr>
<td>EDSMModel</td>
<td>class in hyperspy.models.edstemmodel</td>
<td>410</td>
</tr>
<tr>
<td>EDSMParametersUI</td>
<td>class in hyperspy_signals.eds</td>
<td>231</td>
</tr>
<tr>
<td>EDSM Spectrum</td>
<td>class in hyperspy_signals.eds</td>
<td>232</td>
</tr>
<tr>
<td>edx_cross_section_to_zeta()</td>
<td>(in module hyperspy.misc eds.utils)</td>
<td>372</td>
</tr>
<tr>
<td>EDX Spectrum</td>
<td>class in hyperspy.io_plugins.bruker</td>
<td>339</td>
</tr>
<tr>
<td>EELSConstant()</td>
<td>(in module hyperspy.misc.eels.tools)</td>
<td>379</td>
</tr>
<tr>
<td>EELSCLEdge</td>
<td>class in hyperspy_components.eels_cl_edge</td>
<td>194</td>
</tr>
<tr>
<td>EELSConfig</td>
<td>(in module hyperspy.defaults_parser)</td>
<td>472</td>
</tr>
<tr>
<td>eelsdb()</td>
<td>(in module hyperspy.misc.eels.eelsdb)</td>
<td>376</td>
</tr>
<tr>
<td>EELSModel</td>
<td>class in hyperspy.models.eelsmodel</td>
<td>411</td>
</tr>
<tr>
<td>EELSSpectrum</td>
<td>class in hyperspy_signals.eels</td>
<td>236</td>
</tr>
<tr>
<td>EELSSpectrum_mixin</td>
<td>class in hyperspy_signals.eels</td>
<td>236</td>
</tr>
<tr>
<td>EELSTEMParametersUI</td>
<td>class in hyperspy_signals.eels</td>
<td>244</td>
</tr>
<tr>
<td>effective_angle</td>
<td>(hyperspy_components.eels_cl_edge.EELSCLEdge attribute)</td>
<td>195</td>
</tr>
<tr>
<td>effective_angle()</td>
<td>(in module hyperspy_signals.eels)</td>
<td>377</td>
</tr>
<tr>
<td>effective_Z</td>
<td>(in module hyperspy_misc.eels.effective_angle)</td>
<td>377</td>
</tr>
<tr>
<td>electron_range()</td>
<td>(in module hyperspy_misc.physics_tools)</td>
<td>395</td>
</tr>
<tr>
<td>emi_reader()</td>
<td>(in module hyperspy_io_plugins.fei)</td>
<td>372</td>
</tr>
<tr>
<td>emixml2dtb()</td>
<td>(in module hyperspy_io_plugins.fei)</td>
<td>355</td>
</tr>
<tr>
<td>enable_adjust_position()</td>
<td>hyperspy_models.model1d.Model1D method</td>
<td>419</td>
</tr>
<tr>
<td>enable_adjust_position()</td>
<td>hyperspy_models.model2d.Model2D method</td>
<td>423</td>
</tr>
<tr>
<td>enable_background()</td>
<td>hyperspy_models.eelsmodel.EELSModel method</td>
<td>412</td>
</tr>
<tr>
<td>enable_edges()</td>
<td>hyperspy_models.eelsmodel.EELSModel method</td>
<td>412</td>
</tr>
<tr>
<td>enable_free_onset_energy()</td>
<td>hyperspy_models.eelsmodel.EELSModel method</td>
<td>412</td>
</tr>
<tr>
<td>enable_fine_structure()</td>
<td>hyperspy_models.eelsmodel.EELSModel method</td>
<td>412</td>
</tr>
</tbody>
</table>

**E**

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Class/Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0()</td>
<td>hyperspy_components.eels_cl_edge.EELSCLEdge property</td>
<td>195</td>
</tr>
<tr>
<td>EDS_mixin</td>
<td>class in hyperspy_signals.eds</td>
<td>221</td>
</tr>
<tr>
<td>EDSConfig</td>
<td>class in hyperspy.defaults_parser</td>
<td>472</td>
</tr>
<tr>
<td>EDSModel</td>
<td>class in hyperspy.models.edstemmodel</td>
<td>406</td>
</tr>
<tr>
<td>EDSSEM_mixin</td>
<td>class in hyperspy_signals.eds_sem</td>
<td>230</td>
</tr>
<tr>
<td>EDSSEMModel</td>
<td>class in hyperspy_models.edssemmodel</td>
<td>410</td>
</tr>
<tr>
<td>EDSSEMParametersUI</td>
<td>class in hyperspy_signals.eds_sem</td>
<td>229</td>
</tr>
<tr>
<td>EDSSEM Spectrum</td>
<td>class in hyperspy_signals.eds_sem</td>
<td>230</td>
</tr>
<tr>
<td>EDSSpectrum</td>
<td>class in hyperspy_signals.eds</td>
<td>221</td>
</tr>
</tbody>
</table>
spy.models.eelsmodel.EELSMomodel method), 340
enable_plot_components() (hyper-
spy.model.BaseModel method), 485
enable_plot_components() (hyper-
spy.models.model1d.Model1D
method), 420
enable_xray_lines() (hyper-
spy.models.edsmode1.EDSMomodel
method), 408
energy_axis (hyper-
spy.misc.eels.hartree_slater_gos.HartreeSlaterGOS
attribute), 378
energy_axis (hyper-
spy.misc.eels.hydrogenic_gos.HydrogenicGOS
attribute), 379
energy_onset (hyper-
spy.misc.eels.hartree_slater_gos.HartreeSlaterGOS
attribute), 378
energy_onset (hyper-
spy.misc.eels.hydrogenic_gos.HydrogenicGOS
attribute), 379
energy_to_channel() (hyper-
spy.io_plugins.bruker.EDXSpectrum method), 340
enorm() (hyperspy.external.mpfit.mpfit.mpfit method), 313
ensure_directory() (in module hyper-
spy.misc.io.tools), 384
ensure_parameters_in_bounds() (hyper-
spy.model.BaseModel method), 485
ensure_unicode() (in module hyperspy.misc.utils), 402
epics_metadata (hyperspy.external.tifffile.TiffFile attribute), 325
episc_tags (hyperspy.external.tifffile.TiffPage attribute), 333
Erf (class in hyperspy._components.error_function), 197
estimate_background_windows() (hyper-
spy_signals.eds.EDS_mixin method), 222
estimate_elastic_scattering_intensity() (hyper-
spy_signals.eels.EELSSpectrum_mixin
method), 238
estimate_elastic_scattering_threshold() (hyper-
spy_signals.eels.EELSSpectrum_mixin
method), 238
estimate_fringe_contrast_fourier() (in module hyperspy.misc.holography.tools), 382
estimate_image_shift() (in module hyper-
spy_signals.signal2d), 267
estimate_integration_windows() (hyper-
spy_signals.eds.EDS_mixin method), 223
estimate_map_channels() (hyper-
spy.io_plugins.bruker.HyperHeader method),
estimate_map_depth() (hyper-
spy.io_plugins.bruker.HyperHeader method), 340
estimate_parameters() (hyper-
spy_components.gaussian.Gaussian method), 200
estimate_parameters() (hyper-
spy_components.gaussianhf.GaussianHF
method), 202
estimate_parameters() (hyper-
spy_components.offset.Offset method), 204
estimate_parameters() (hyper-
spy_components.polynomial.Polynomial
method), 206
estimate_parameters() (hyper-
spy_components.polynomial Deprecated.Polynomial
method), 207
estimate_parameters() (hyper-
spy_components.power_law.PowerLaw
method), 208
estimate_parameters() (hyper-
spy_components.voigt.Voigt method), 212
estimate_peak_width() (hyper-
spy_signals.signal1d.Signal1D
method), 256
estimate_poissonian_noise_variance() (hyper-
spy.signal.BaseSignal method), 507
estimate_shift1D() (hyper-
spy_signals.signal1d.Signal1D
method), 256
estimate_shift2D() (hyper-
spy_signals.signal2d.Signal2D
method), 265
estimate_sideband_position() (hyper-
spy_signals.hologram_image.HologramImage
method), 244
estimate_sideband_position() (in module hyper-
spy.missc.holography.reconstruct), 381
estimate_sideband_size() (hyper-
spy_signals.hologram_image.HologramImage
method), 245
estimate_sideband_size() (in module hyper-
spy.misc.holography.reconstruct), 381
estimate_thickness() (hyper-
spy_signals.eels.EELSSpectrum_mixin
method), 239
estimate_variance_parameters() (in module hyperspy.misc.eels.tools), 380
estimate_zero_loss_peak_centre() (hyper-
spy_signals.eels.EELSSpectrum_mixin
method), 239
eval() (hyperspy.external.astroML.histtools.KnuthF
method), 301
<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event (class in hyperspy.events)</td>
<td></td>
<td>473</td>
</tr>
<tr>
<td>Events (class in hyperspy.events)</td>
<td></td>
<td>476</td>
</tr>
<tr>
<td>Events (class in hyperspy.external.astroML.bayesian_blocks)</td>
<td></td>
<td>298</td>
</tr>
<tr>
<td>EventSuppressor (class in hyperspy.events)</td>
<td></td>
<td>475</td>
</tr>
<tr>
<td>exp_decay() (in module hyperspy.samfire.utils.local_strategies)</td>
<td></td>
<td>425</td>
</tr>
<tr>
<td>explained_variance</td>
<td>hyperspy.learn.mva.LearningResults attribute</td>
<td>363</td>
</tr>
<tr>
<td>explained_variance_ratio</td>
<td>hyperspy.learn.mva.LearningResults attribute</td>
<td>363</td>
</tr>
<tr>
<td>Exponential (class in hyperspy.components.exponential)</td>
<td></td>
<td>198</td>
</tr>
<tr>
<td>export() (hyperspy.component.Component method)</td>
<td></td>
<td>466</td>
</tr>
<tr>
<td>export() (hyperspy.component.Parameter method)</td>
<td></td>
<td>469</td>
</tr>
<tr>
<td>export() (hyperspy.misc.utils.DictionaryTreeBrowser method)</td>
<td></td>
<td>400</td>
</tr>
<tr>
<td>export_bss_results() (hyperspy.signal.MVATools method)</td>
<td></td>
<td>527</td>
</tr>
<tr>
<td>export_decomposition_results()</td>
<td>hyperspy.signal.MVATools method</td>
<td>528</td>
</tr>
<tr>
<td>export_results() (hyperspy.model.BaseModel method)</td>
<td></td>
<td>483</td>
</tr>
<tr>
<td>export_results() (hyperspy.models.model1d.Model1D method)</td>
<td></td>
<td>418</td>
</tr>
<tr>
<td>export_to_dictionary() (in module hyperspy.misc.export_dictionary)</td>
<td></td>
<td>390</td>
</tr>
<tr>
<td>Expression (class in hyperspy.components.expression)</td>
<td></td>
<td>198</td>
</tr>
<tr>
<td>ext_bounded (hyperspy.component.Parameter attribute)</td>
<td></td>
<td>469</td>
</tr>
<tr>
<td>ext_bounded() (hyperspy.component.Parameter property)</td>
<td></td>
<td>471</td>
</tr>
<tr>
<td>ext_force_positive (hyperspy.component.Parameter attribute)</td>
<td></td>
<td>469</td>
</tr>
<tr>
<td>ext_force_positive() (hyperspy.component.Parameter property)</td>
<td></td>
<td>471</td>
</tr>
<tr>
<td>extend() (hyperspy.model.BaseModel method)</td>
<td></td>
<td>482</td>
</tr>
<tr>
<td>extend() (hyperspy.models.model1d.Model1D method)</td>
<td></td>
<td>417</td>
</tr>
<tr>
<td>extend() (hyperspy.models.model2d.Model2D method)</td>
<td></td>
<td>422</td>
</tr>
<tr>
<td>extend() (hyperspy.samfire.Samfire method)</td>
<td></td>
<td>499</td>
</tr>
<tr>
<td>extend() (hyperspy.samfire.StrategyList method)</td>
<td></td>
<td>501</td>
</tr>
<tr>
<td>extrasamples (hyperspy.external.tifffile.TiffPage attribute)</td>
<td></td>
<td>333</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factors (hyperspy.learn.mva.LearningResults attribute)</td>
<td></td>
<td>363</td>
</tr>
<tr>
<td>FancySlicing (class in hyperspy.misc.slicing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fei_check() (in module hyperspy.io_plugins.emd)</td>
<td></td>
<td>355</td>
</tr>
<tr>
<td>fei_metadata (hyperspy.external.tifffile.TiffFile attribute)</td>
<td></td>
<td>325</td>
</tr>
<tr>
<td>FeiEMDReader (class in hyperspy.io_plugins.emd)</td>
<td></td>
<td>354</td>
</tr>
<tr>
<td>FeiSpectrumStream (class in hyperspy.io_plugins.emd)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft() (hyperspy.signal.BaseSignal method)</td>
<td></td>
<td>508</td>
</tr>
<tr>
<td>fft_correlation() (in module hyperspy.signals.signal2d)</td>
<td></td>
<td>268</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.blockfile)</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.bruker)</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.dens)</td>
<td></td>
<td>343</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.digital_micrograph)</td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.edax)</td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.feix)</td>
<td></td>
<td>355</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.mrc)</td>
<td></td>
<td>357</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.msa)</td>
<td></td>
<td>357</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.netcdf)</td>
<td></td>
<td>357</td>
</tr>
<tr>
<td>file_reader() (in module hyperspy.io_plugins.writers)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index
<table>
<thead>
<tr>
<th>Index</th>
<th>HyperSpy Documentation, Release 1.5.1.dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>233</td>
<td>get_calibration_from_position() (in module hyperspy.io_plugins.fei), 355</td>
</tr>
<tr>
<td></td>
<td>get_centre() (hyperspy.drawing._widgets.circle.CircleWidget method), 278</td>
</tr>
<tr>
<td></td>
<td>get_centre() (hyperspy.drawing._widgets.line2d.Line2DWidget method), 279</td>
</tr>
<tr>
<td></td>
<td>get_centre() (hyperspy.drawing.widget.ResizableDraggableWidgetBase method), 295</td>
</tr>
<tr>
<td></td>
<td>get_centre_index() (hyperspy.drawing.widget.ResizableDraggableWidget method), 295</td>
</tr>
<tr>
<td></td>
<td>get_complex() (hyperspy.drawing.signal1d.Signal1DLine property), 287</td>
</tr>
<tr>
<td></td>
<td>get_configuration_directory_path() (in module hyperspy.api_nogui), 461</td>
</tr>
<tr>
<td></td>
<td>get_core_loss_eels_line_scan_signal() (in module hyperspy.datasets.artificial_data), 269</td>
</tr>
<tr>
<td></td>
<td>get_core_loss_eels_model() (in module hyperspy.datasets.artificial_data), 269</td>
</tr>
<tr>
<td></td>
<td>get_core_loss_eels_signal() (in module hyperspy.datasets.artificial_data), 270</td>
</tr>
<tr>
<td></td>
<td>get_current_signal() (hyperspy.signal.BaseSignal method), 509</td>
</tr>
<tr>
<td></td>
<td>get_data() (hyperspy.io_plugins.digital_micrograph.ImageObject), 344</td>
</tr>
<tr>
<td></td>
<td>get_data_dtype_list() (in module hyperspy.io_plugins.fei), 355</td>
</tr>
<tr>
<td></td>
<td>get_data_position() (hyperspy.drawing.marker.MarkerBase method), 284</td>
</tr>
<tr>
<td></td>
<td>get_data_reader() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 343</td>
</tr>
<tr>
<td></td>
<td>get_data_tag_dtype_list() (in module hyperspy.io_plugins.fei), 355</td>
</tr>
<tr>
<td></td>
<td>get_data_type() (in module hyperspy.io_plugins.mrc), 357</td>
</tr>
<tr>
<td></td>
<td>get_date_time_from_header() (in module hyperspy.misc.date_time_tools), 388</td>
</tr>
<tr>
<td></td>
<td>get_decomposition_factors() (hyperspy.signal.MVATools method), 529</td>
</tr>
<tr>
<td></td>
<td>get_decomposition_loadings() (hyperspy.signal.MVATools method), 529</td>
</tr>
<tr>
<td></td>
<td>get_decomposition_model() (hyperspy.learn.mva.MVA method), 365</td>
</tr>
<tr>
<td></td>
<td>get_default_header() (in module hyperspy.io_plugins.mrc), 357</td>
</tr>
<tr>
<td></td>
<td>get_derivative() (in module hyperspy.learn.mva), 368</td>
</tr>
<tr>
<td></td>
<td>get_dictionary() (hyperspy.io_plugins.protochips.ProtochipsCSV method), 357</td>
</tr>
<tr>
<td></td>
<td>get_dimensions_from_data() (hyperspy.signal.BaseSignal method), 509</td>
</tr>
<tr>
<td></td>
<td>get_electron_energy_loss_spectrum() (hyperSpy_signals.dielectric_function.DielectricFunctionMixin method), 220</td>
</tr>
<tr>
<td></td>
<td>get_explained_variance_ratio() (hyperspy.learn.mva.MVA method), 365</td>
</tr>
<tr>
<td></td>
<td>get_fei_dtype_list() (in module hyperspy.io_plugins.fei), 355</td>
</tr>
<tr>
<td></td>
<td>get_file() (hyperspy.io_plugins.bruker.SFS_reader method), 341</td>
</tr>
<tr>
<td></td>
<td>get_fine_structure_as_signal1D() (hyperSpy_components.eels.cl_edge.EELSCLEdge method), 195</td>
</tr>
<tr>
<td></td>
<td>get_FWHM_at_Energy() (hyperSpy.misc.eeds.utils), 373</td>
</tr>
<tr>
<td></td>
<td>get_gui() (in module hyperspy.ui_registry), 539</td>
</tr>
<tr>
<td></td>
<td>get_header_dtype_list() (in module hyperspy.io_plugins.blockfile), 339</td>
</tr>
<tr>
<td></td>
<td>get_header_dtype_list() (in module hyperspy.io_plugins.fei), 355</td>
</tr>
<tr>
<td></td>
<td>get_header_from_signal() (in module hyperspy.io_plugins.fei), 339</td>
</tr>
<tr>
<td></td>
<td>get_histogram() (hyperSpy_signals.lazy.LazySignal method), 250</td>
</tr>
<tr>
<td></td>
<td>get_histogram() (hyperspy.signal.BaseSignal method), 509</td>
</tr>
<tr>
<td></td>
<td>get_hspy_format_version() (in module hyperspy.io_plugins.hspy), 356</td>
</tr>
<tr>
<td></td>
<td>get_image_dictionaries() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 343</td>
</tr>
<tr>
<td></td>
<td>get_interactive_ns() (in module hyperspy.misc.ipython_tools), 391</td>
</tr>
<tr>
<td></td>
<td>get_interpolated_spectrum() (hyperspy.signal_tools.SpikesRemoval method), 538</td>
</tr>
<tr>
<td></td>
<td>get_interpolation_range() (hyperspy.signal_tools.SpikesRemoval method), 538</td>
</tr>
<tr>
<td></td>
<td>get_ipr_dtype_list() (in module hyperspy.io_plugins.edax), 345</td>
</tr>
<tr>
<td></td>
<td>get_ipython() (in module hyperspy.misc.ipython_tools), 391</td>
</tr>
<tr>
<td></td>
<td>get_item() (hyperspy.misc.ipython_tools.DictionaryTreeBrowser method), 400</td>
</tr>
<tr>
<td></td>
<td>get_iter_and_properties() (hyperspy.io_plugins.braker.SFSTreeItem method), 341</td>
</tr>
<tr>
<td></td>
<td>get_lengths() (in module hyperspy.io_plugins.fei), 339</td>
</tr>
</tbody>
</table>
### grad_ab() (hyperspy_components.pes_core_line_shape.PESCLineShape, 205)

### grad_b() (hyperspy_components.bleasdale.Bleasdale, 194)

### grad_B() (hyperspy_components.pes_see.SEE, 205)

### grad_c() (hyperspy_components.bleasdale.Bleasdale, 194)

### grad_coefficients() (hyperspy_components.polynomial_deprecated.Polynomial, 207)

### grad_FWHM() (hyperspy_components.pes_core_line_shape.PESCLineShape, 205)

### grad_intensity() (hyperspy_components.eels_cl_edge.EELSCLEdge, 195)

### grad_intensity() (hyperspy_components.volume_plasmon_drude.VolumePlasmonDrude, 209)

### grad_k() (hyperspy_components.arctan.Arctan, 193)

### grad_n() (hyperspy_components.heaviside.HeavisideStep, 203)

### grad_offset() (hyperspy_components.offset.Offset, 205)

### grad_one_coefficient() (hyperspy_components.polynomial_deprecated.Polynomial, 207)

### grad_origin() (hyperspy_components.eels_double_power_law.DoublePowerLaw, 197)

### grad_origin() (hyperspy_components.pes_core_line_shape.PESCLineShape, 205)

### grad_origin() (hyperspy_components.power_law.PowerLaw, 208)

### grad_Phi() (hyperspy_components.pes_see.SEE, 205)

### grad_plasmon_energy() (hyperspy_components.volume_plasmon_drude.VolumePlasmonDrude, 213)

### grad_r() (hyperspy_components.eels_double_power_law.DoublePowerLaw, 197)

### grad_r() (hyperspy_components.power_law.PowerLaw, 208)

### grad_ratio() (hyperspy_components.eels_double_power_law.DoublePowerLaw, 197)

### grad_shift() (hyperspy_components.eels_double_power_law.DoublePowerLaw, 205)
property), 295
init_parameters() (hyperspy.component.Component method), 467
insert() (hyperspy.model.BaseModel method), 487
integral_as_signal() (hyperspy_components.gaussianhf.GaussianHF method), 202
integrate1D() (hyperspy.signal.BaseSignal method), 512
integrate_in_range() (hyperspy_signals.signal1d.Signal1D method), 258
integrate_simpson() (hyperspy_signals.lazy.LazySignal method), 251
IntegrateArea (class in hyperspy.signal_tools), 536
integrateq() (hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS_fluoview() (hyperspy.signal.signal1d.BaseSignal method), 512
interpolate1D() (in module hyperspy._signals.signal1d), 263
interpolate_in_between() (hyperspy_signals.signal1d.Signal1D method), 259
interpolator (hyperspy.signal_tools.SpikesRemoval attribute), 538
interpret() (in module hyperspy.io_plugins.braker), 342
ipython_kwars (hyperspy.samfire_utils.samfire_pool.SamfirePool attribute), 426
ipython_kwars (hyperipython-utils.parallel_pool.ParallelPool attribute), 456
is_active() (hyperspy.drawing.mpl_he.MPL_HyperExplorer property), 285
is_andor() (hyperspy.external.tifffile.TiffPage property), 333
is_bigtiff() (hyperspy.external.tifffile.TiffFile property), 325
is_chroma_subsampled() (hyperspy.external.tifffile.TiffPage property), 333
is_contiguous (hyperspy.external.tifffile.TiffPage attribute), 333
is_contiguous() (hyperspy.external.tifffile.TiffPage attribute), 335
is_eels() (hyperspy.external.tifffile.TiffPage property), 333
is_fei() (hyperspy.external.tifffile.TiffPage property), 333
is_file (hyperspy.external.tifffile.FileHandle attribute), 330
is_final (hyperspy.external.tifffile.TiffPage attribute), 333
is_flag (hyperspy.external.tifffile.TiffFile attribute), 323
is_geotiff() (hyperspy.external.tifffile.TiffPage property), 333
is_imagej (hyperspy.external.tifffile.TiffPage attribute), 333
is_ipyparallel (hyperspy.utils.parallel_pool.ParallelPool attribute), 457
is_ipyparallel() (hyperspy.utils.parallel_pool.ParallelPool property), 457
is_it_running_from_ipython() (in module hyperspy.misc.ipython_tools), 391
is_lsm() (hyperspy.external.tifffile.TiffPage property), 333
is_mask() (hyperspy.external.tifffile.TiffPage property), 333
is_mdgel (hyperspy.external.tifffile.TiffFile attribute), 325
is_mdgel() (hyperspy.external.tifffile.TiffPage attribute), 335
is_mdgel() (hyperspy.external.tifffile.TiffPage property), 333
is_mediacy () (hyperspy.external.tifffile.TiffPage property), 333
is_memmappable (hyperspy.external.tifffile.TiffPage attribute), 333
is_memmappable() (hyperspy.external.tifffile.TiffFrame property), 336
is_memmappable() (hyperspy.external.tifffile.TiffFrame attribute), 333
is_movie() (hyperspy.external.tifffile.TiffFile property), 325
is_mrc() (hyperspy.external.tifffile.TiffPage property), 334
is_multipage() (hyperspy.external.tifffile.TiffPage property), 334
is_multiprocessing (hyperspy.utils.parallel_pool.ParallelPool property), 457
is_multiprocessing() (hyperspy.utils.parallel_pool.ParallelPool property), 457
is_ndpi (hyperspy.external.tifffile.TiffPage property), 334
is_nih() (hyperspy.external.tifffile.TiffPage property), 334
is_ome() (hyperspy.external.tifffile.TiffPage property), 334
is_on() (hyperspy.drawing.widget.WidgetBase method), 297
is_pilatus() (hyperspy.external.tifffile.TiffPage property), 334
is_qptiff() (hyperspy.external.tifffile.TiffPage property), 334
is_reduced() (hyperspy.external.tifffile.TiffPage property), 334
is_rgb() (hyperspy.signal.BaseSignal property), 513
is_rgb() (in module hyperspy.misc.rgb_tools), 395
is_rgb() (hyperspy.signal.BaseSignal property), 513
is_rgb() (in module hyperspy.misc.rgb_tools), 395
is_rgbx() (hyperspy.signal.BaseSignal property), 513
is_rgbx() (in module hyperspy.misc.rgb_tools), 395
is_scanimage() (hyperspy.external.tifffile.TiffPage property), 334
is_scn() (hyperspy.external.tifffile.TiffPage property), 334
is_sem() (hyperspy.external.tifffile.TiffPage property), 334
is_sgi() (hyperspy.external.tifffile.TiffPage property), 334
is_shaped (hyperspy.external.tifffile.TiffPage attribute), 334
is_sis() (hyperspy.external.tifffile.TiffPage property), 334
is_stk() (hyperspy.external.tifffile.TiffPage property), 334
is_svs() (hyperspy.external.tifffile.TiffPage property), 334
is_tiled() (hyperspy.external.tifffile.TiffPage property), 334
is_tvips() (hyperspy.external.tifffile.TiffPage property), 334
is_valid() (hyperspy.roi.Point2DROI method), 496
is_valid() (hyperspy.roi.RectangularROI method), 497
is_valid() (hyperspy.roi.SemperROI method), 497
is_vista() (hyperspy.external.tifffile.TiffPage property), 334
is_valid() (hyperspy.misc.math_tools.isiterable()) (hyperspy.drawing.marker.MarkerBase method), 284
is_valid() (in module hyperspy.misc.utils), 403
is_valid() (in module hyperspy.misc.date_time_tools), 387
iterable_not_string() (in module hyperspy.misc.utils), 403
isfloat() (in module hyperspy.misc.math_tools), 394
isiterable() (hyperspy.misc.date_time_tools.KnuthF), 394
isiterable() (in module hyperspy.misc.date_time_tools), 387

J
jit_ifnumba() (in module hyperspy.decorators), 472

K
key2switch_right_pointer() (hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer method), 286
key_navigator() (hyperspy.axes.AxesManager method), 464
key_press_handler_custom() (in module hyperspy.drawing.utils), 288
keyframe (hyperspy.external.tifffile.TiffFrame attribute), 336
keyframe() (hyperspy.external.tifffile.TiffPage property), 334
keys() (hyperspy.misc.utils.DictionaryTreeBrowser method), 401
knuth_bin_width() (in module hyperspy.external.astroML.histtools), 303
KnuthF (class in hyperspy.external.astroML.histtools), 300
kramers_kronig_analysis() (hyperspy_signals.eels.EELSSpectrumMixin method), 241

L
LABEL_DTYPES (hyperspy.io_plugins.semper_unf.SemperFormat attribute), 360
LabelWidget (class in hyperspy.drawing_widgets.label), 279
lazify() (hyperspy.tests.model.test_chi_squared.TestChiSquared method), 439
lazify() (hyperspy.tests.model.test_fancy_indexing.TestEELSModelSlicing method), 440
lazify() (hyperspy.tests.model.test_fancy_indexing.TestModelIndexing method), 440
<table>
<thead>
<tr>
<th>Method</th>
<th>Class/Module/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.model.test_fancy_indexing.TestModelIndexingClass() (in hyperspy.signals.eels), 244</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_sem.Test_energy_loss.SpecimenHologramImage (class in hyperspy.signals.hologram_image), 447</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_sem.Test_get_line_spectra_in_image LazySignal1D (class in hyperspy.signals.signal1d), 248</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_sem.Test_get_metadata LazySignal2D (class in hyperspy.signals.signal2d), 254</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_sem.Test_tools_bulk LearningResults (class in hyperspy.learn.mva), 264</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_tem.Test_metadata &lt;g&gt;legend_artist() (hyperspy.signal_tools.DerivativeTextHandler method), 494</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_tem.Test_quantification_length() (hyperspy.roi.Line2DROI property), 535</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_eds_tem.Test_simple_model_line_properties() (hyperspy.signal_tools.Line2DROI property), 495</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_image.Test2DLineInSignal1D (class in hyperspy.signal1d), 313</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_image.Test3DLineSegment (class in hyperspy.signal1d), 536</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_image.Test4DLine properties (hyperspy.signal1d.Signal1DLine attribute), 536</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_spectrum.Test2DLineInSignal1D (class in hyperspy.signal1d), 472</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_spectrum.Test3DLineSegment (class in hyperspy.signal1d), 487</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.tests.signal.test_spectrum.Test4DLine properties (hyperspy.signal1d.Signal1DLine attribute), 495</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.misc.example_signals_loading.load_1D_EDS_SEM_spectrum() (in hyperspy.misc.example_signals_loading), 389</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.misc.example_signals_loading.load_1D_EDS_TEM_spectrum() (in hyperspy.misc.example_signals_loading), 389</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.misc.example_signals_loading.load_from_dict() (hyperspy.misc.example_signals_loading), 389</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.misc.example_signals_loading.load_from_emd() (hyperspy.io_plugins.emd.EMD class method), 353</td>
</tr>
<tr>
<td>lazify()</td>
<td>hyperspy.io_plugins.fei.load_only_data() (in hyperspy.io_plugins.fei), 472</td>
</tr>
<tr>
<td>lazyattr</td>
<td>hyperspy.io_plugins.fei.load_1D_EDS_SEM_spectrum() (in hyperspy.io), 478</td>
</tr>
<tr>
<td>LazyComplexSignal</td>
<td>hyperspy.misc.example_signals.load_parameters_from_file() (hyperspy.model.BaseModel method), 487</td>
</tr>
<tr>
<td>LazyComplexSignal1D</td>
<td>hyperspy.misc.example_signals.load_1D_EDS_SEM_spectrum() (in hyperspy.model), 389</td>
</tr>
<tr>
<td>LazyComplexSignal2D</td>
<td>hyperspy.misc.example_signals.load_1D_EDS_TEM_spectrum() (in hyperspy.model), 389</td>
</tr>
<tr>
<td>LazyDielectricFunction</td>
<td>hyperspy.misc.example_signals.load_from_dict() (hyperspy.misc.example_signals_loading), 389</td>
</tr>
<tr>
<td>LazyEDSSEMSpectrum</td>
<td>hyperspy.misc.example_signals.load_from_emd() (hyperspy.io_plugins.emd.EMD class method), 353</td>
</tr>
<tr>
<td>LazyEDSSpectrum</td>
<td>hyperspy.misc.example_signals.load_only_data() (in hyperspy.io), 389</td>
</tr>
<tr>
<td>LazyEDSTEMSpectrum</td>
<td>hyperspy.misc.example_signals.load_from_unf() (hyperspy.io_plugins.semper_unf.SemperFormat class method), 360</td>
</tr>
<tr>
<td>LazyHologramImage</td>
<td>hyperspy.misc.example_signals.load_object_hologram() (in hyperspy.misc.example_signals_loading), 389</td>
</tr>
<tr>
<td>LazyEDSTEMSpectrum</td>
<td>hyperspy.misc.example_signals.load_from_unf() (hyperspy.io_plugins.semper_unf.SemperFormat class method), 360</td>
</tr>
</tbody>
</table>
HyperSpy Documentation, Release 1.5.1.dev

Project() (hyperspy.learn.rpc.a.ORPCA method), 369
ProtochipsCSV (class in hyperspy.io_plugins.protochips), 357
py_parse_hypermap() (in module hyperspy.io_plugins.bruker), 342

Q
qaxis (hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS attribute), 378
qaxis (hyperspy.misc.eels.hydrogenic_gos.HydrogenicGOS attribute), 379
qptiff_metadata (hyperspy.external.tifffile.TiffFile attribute), 325
qrfac() (hyperspy.external.mpfit.mpfit.mpfit method), 313
qrsolv() (hyperspy.external.mpfit.mpfit.mpfit method), 313
quantification() (hyper- spy_signals.eds_tem.EDSTEM_mixin method), 233
quantification_cliff_lorimer() (in module hyperspy.misc.eels.edsutils), 373
quantification_cross_section() (in module hyperspy.misc.eels.edsutils), 374
quantification_zeta_factor() (in module hyperspy.misc.eels.edsutils), 374
quantify() (hyperspy.models.eelsmodel.EELSModel method), 415

R
radial() (hyperspy.samfire_utils.strategy.LocalStrategy property), 429
range() (hyperspy.drawing_widgets.range.ModifiableSpanSelector property), 280
RangeWidget (class in hyperspy_components), 208
read() (hyperspy.external.tifffile.FileHandle method), 330
read_array() (hyperspy.external.tifffile.FileHandle method), 330
read_array() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 344
read_boolean() (in module hyperspy.misc.io.utils_readfile), 384
read_byte() (in module hyperspy.misc.io.utils_readfile), 384
read_char() (in module hyperspy.misc.io.utils_readfile), 384
read_double() (in module hyperspy.misc.io.utils_readfile), 384
read_elements() (hyperspy.misc.eels.base_gos.GOSBase method), 376
read_float() (in module hyperspy.misc.io.utils_readfile), 385
read_l_or_q() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader.property), 344
read_long() (in module hyperspy.misc.io.utils_readfile), 385
read_long_long() (in module hyperspy.misc.io.utils_readfile), 385
read_piece() (hyperspy.io_plugins.bruker.SFSTreeItem method), 341
read_raw() (in module hyperspy.io_plugins.ripple), 359
read_record() (hyperspy.external.tifffile.FileHandle method), 330
read_short() (in module hyperspy.misc.io.utils_readfile), 385
read_simple_data() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 344
read_string() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 344
read_struct() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 344
read_ulong() (in module hyperspy.misc.io.utils_readfile), 385
read_ulong_long() (in module hyperspy.misc.io.utils_readfile), 385
read_ushort() (in module hyperspy.misc.io.utils_readfile), 385
readgosfile() (hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS method), 378
readLELong() (in module hyperspy.io_plugins.fei), 356
readLELongLong() (in module hyperspy.io_plugins.fei), 356
readLEShort() (in module hyperspy.io_plugins.fei), 356
readrealfile() (hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS method), 378
real() (hyperspy._signals.complex_signal.ComplexSignal_mixin property), 216
rebin() (hyperspy.misc.complex_signal.ComplexSignal_mixin method), 227
rebin() (hyperspy.signals.eels.EELSSpectrum_mixin method), 242
rebin() (hyperspy_signals.lazy.LazySignal method), 252
rebin() (hyperspy.signal.BaseSignal method), 517
rebin() (in module hyperspy.misc.array_tools), 386
recompute_out() (hyperspy.interactive.Interactive method), 478
reconstruct() (in module hyperspy.misc.holography.reconstruct), 382
reconstruct_component() (in module hyperspy.model), 492
reconstruct_object() (in module hyperspy.misc.export_dictionary), 391
reconstruct_phase() (hyperspy_signals.hologram_image.HologramImage method), 245
record_by() (hyperspy.io_plugins.digital_micrograph.ImageObject property), 344
Rectangle (class in hyperspy.drawing._markers.rectangle), 275
RectangleWidget (class in hyperspy.drawing._widgets.rectangles), 281
RectangularROI (class in hyperspy.roi), 496
red_chisq (hyperspy.model.BaseModel attribute), 482
red_chisq (hyperspy.models.model1d.Model1D attribute), 417
red_chisq (hyperspy.models.model2d.Model2D attribute), 422
red_chisq() (hyperspy.model.BaseModel property), 489
red_chisq_test (class in hyperspy.samfire_utils.goodness_of_fit_tests.red_chisq), 424
ReducedChiSquaredStrategy (class in hyperspy.samfire_utils.local_strategies), 425
ReducedChiSquaredWeight (class in hyperspy.samfire_utils.weights.red_chisq), 424
refresh() (hyperspy.samfire_utils.strategy.GlobalStrategy method), 428
refresh() (hyperspy.samfire_utils.strategy.LocalStrategy method), 429
refresh_database() (hyperspy.samfire.samfire_worker.Worker method), 499, 500
regular_array2rgbx() (in module hyperspy.misc.rgb_tools), 395
RegularEvents (class in hyperspy.external.astroML.bayesian_blocks), 299
rel_std_of_fraction() (in module hyperspy.misc.eels_tools), 380
release() (hyperspy.drawing._widgets.range.ModifiableSpanSelector method), 538
remove() (hyperspy.models.edsmodel.EDSModel method), 415
remove() (hyperspy.models.model1d.Model1D method), 417, 421
remove() (hyperspy.models.model2d.Model2D method), 422
remove() (hyperspy.samfire.Samfire method), 499, 500
remove() (hyperspy.samfire.StrategyList method), 501
remove() (hyperspy.samfire_utils.strategy.SamfireStrategy method), 430
remove() (hyperspy.signal.ModelManager method), 533
remove_background() (hyperspy._signals.signal1d.Signal1D method), 259
remove_fine_structure_data() (hyperspy.models.edsmodel.EELSMModel method), 415
remove_right_pointer() (hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer method), 286
remove_signal_range() (hyperspy.models.model1d.Model1D method), 421
remove_signal_range() (hyperspy.models.model2d.Model2D method), 423
reset_span_selector() (hyperspy.signal_tools.SignalRangeEditor method), 538
reset_the_signal_range() (hyperspy.models.edsmodel.EDSModel method), 415
reset_the_signal_range() (hyperspy.models.model1d.Model1D method), 421
reset_the_signal_range() (hyperspy.models.model2d.Model2D method), 423
remove() (hyperspy.models.edsmodel.EDSModel method) ResizableDraggableWidgetBase (class in hyperspy._widgets.rectangles), 281
Index 583
send_results() (hyperspy.samfire_utils.samfire_worker.Worker method), 428

ser_reader() (in module hyperspy.io_plugins.fei), 356

serial_date_to_datetime() (in module hyperspy.misc.date_time_tools), 388

serial_date_to_ISO_format() (in module hyperspy.misc.date_time_tools), 388

series (hyperspy.external.tifffile.TiffFile attribute), 323, 326

set_all_edges_intensities_positive() (hyperspy.models.eelsmodel.EELSModel method), 416

set_axes_decor() (in module hyperspy.drawing.utils), 294

set_background_estimator() (hyperspy.signal_tools.BackgroundRemoval method), 535

set_boundaries() (hyperspy.model.BaseModel method), 489

set_bounds() (hyperspy.drawing._widgets.range.RangeWidget method), 281

set_bounds() (hyperspy.drawing._widgets.rectangles.RectangleWidget method), 282

set_color() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 283

set_component_active_value() (hyperspy.model.BaseModel method), 489

set_contrast() (hyperspy.drawing.image.ImagePlot method), 284

set_convolution_axis() (hyperspy.models.model1d.Model1D method), 421

set_current_values_to() (hyperspy.model.BaseModel method), 483

set_current_values_to() (hyperspy.models.model1d.Model1D method), 418

set_data() (hyperspy.drawing.marker.MarkerBase method), 285

set_elements() (hyperspy_signals.e53.s_e53.E53Spectrum_mixin method), 228

set_ibounds() (hyperspy.drawing._widgets.range.RangeWidget method), 281

set_ibounds() (hyperspy.drawing._widgets.rectangles.RectangleWidget method), 282

set_initial() (hyperspy.drawing._widgets.range.ModifiableSpanSelector method), 280

set_item() (hyperspy.miss.UTILS.DictionaryTreeBrowser method), 401

set_length() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 283

set_line_properties() (hyperspy.drawing.signal1d.Signal1DLine method), 287

set_lines() (hyperspy_signals.e53_s_e53.E53Spectrum_mixin method), 228

set_log_level() (in module hyperspy.logger), 481

set_marker_properties() (hyperspy.drawing.marker.MarkerBase method), 285

set_microscope_parameters() (hyperspy_components.e53.cl_edge.E53CLedge method), 196

set_microscope_parameters() (hyperspy_signals.e53_e53.E53Spectrum_mixin method), 230

set_microscope_parameters() (hyperspy_signals.e53_e53.E53Spectrum_mixin method), 234

set_mpfit_parameters_info() (hyperspy.model.BaseModel method), 490

set_mpl_ax() (hyperspy.drawing.marker.WidgetBase method), 297

set_on() (hyperspy.drawing._widgets.range.RangeWidget method), 281

set_on() (hyperspy.drawing.marker.ResizersMixin method), 296

set_on() (hyperspy.drawing.marker.WidgetBase method), 297

set_optional_names() (hyperspy.samfire_utils.samfire_worker.Worker method), 285

set_parameter_boundaries() (hyperspy.samfire_utils.samfire_worker.Worker method), 428

set_parameters_free() (hyperspy.component.Component method), 467

set_parameters_free() (hyperspy.model.BaseModel method), 490

set_parameters_not_free() (hyperspy.component.Component method), 468

set_parameters_not_free() (hyperspy.model.BaseModel method), 490
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
<th>Module</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_parameters_value()</td>
<td>(hyperSpy.model.BaseModel method)</td>
<td>483, 491</td>
<td></td>
</tr>
<tr>
<td>set_parameters_value()</td>
<td>(hyperSpy.models.model1d.Model1D method)</td>
<td>418</td>
<td></td>
</tr>
<tr>
<td>set_quantity_label()</td>
<td>(hyperSpy.drawing.image.ImagePlot method)</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>set_signal_dimension()</td>
<td>(hyperSpy.axes.AxesManager method)</td>
<td>464</td>
<td></td>
</tr>
<tr>
<td>set_signal_range()</td>
<td>(hyperSpy.models.model1d.Model1D method)</td>
<td>421</td>
<td></td>
</tr>
<tr>
<td>set_signal_range()</td>
<td>(hyperSpy.models.model2d.Model2D method)</td>
<td>423</td>
<td></td>
</tr>
<tr>
<td>set_signal_type()</td>
<td>(hyperSpy.signal.BaseSignal method)</td>
<td>520</td>
<td></td>
</tr>
<tr>
<td>set_size_in_indices()</td>
<td>(hyperSpy.drawing.widgets.ResizeableDraggableWidgetBase method)</td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>set_tex_bold()</td>
<td>(hyperSpy.drawing._widgets.scalebar.ScaleBar method)</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>set_values()</td>
<td>(hyperSpy.samfire_utils.samfire_worker.Worker method)</td>
<td>428</td>
<td></td>
</tr>
<tr>
<td>set_xaxis_lims()</td>
<td>(in module hyperSpy.drawing.utilities)</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>setup()</td>
<td>(hyperSpy.samfire_utils.samfire_pool.SamfirePool method)</td>
<td>426</td>
<td></td>
</tr>
<tr>
<td>setup()</td>
<td>(hyperSpy.utils.parallel_pool.ParallelPool method)</td>
<td>456, 457</td>
<td></td>
</tr>
<tr>
<td>setup_compression_metadata()</td>
<td>(hyperSpy.io_plugins.bruker.SFSTreeItem method)</td>
<td>341</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.axes.test_axes_manager.TestAxesHotkeys method)</td>
<td>431</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.axes.test_axes_manager.TestAxesManager method)</td>
<td>431</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_component.TestCallMethods method)</td>
<td>432</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_component.TestGeneralMethods method)</td>
<td>432</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_component.TestMultidimensionalActiveArray method)</td>
<td>432</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_component_active_array.TestParametersAsSignals method)</td>
<td>433</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_component_set_parameters.TestSetParameters method)</td>
<td>433</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_components2D.TestExpression2D method)</td>
<td>433</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.component.test_components2D.TestGaussian2D method)</td>
<td>433</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.io.test_msa.TestExample1 method)</td>
<td>437</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.io.test_msa.TestExample1WrongDate method)</td>
<td>437</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.io.test_msa.TestExample2 method)</td>
<td>437</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.misc.test_rgbtools.TestRGBTools method)</td>
<td>438</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_chi_squared.TestChiSquared method)</td>
<td>439</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_fancy_indexing.TestEELSModelSlicing method)</td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_fancy_indexing.TestModelIndexing method)</td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_fancy_indexing.TestModelIndexingClass method)</td>
<td>440</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_model_selection_criteria.TestModelSelection method)</td>
<td>441</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_set_parameter_state.TestSetParameterInModel method)</td>
<td>441</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.model.test_set_parameter_state.TestSetParameterInModel method)</td>
<td>442</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.samfire.test_goodness_of_fit_tests.TestInformationCriteria method)</td>
<td>443</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.samfire.test_goodness_of_fit_tests.TestRedChisq method)</td>
<td>443</td>
<td></td>
</tr>
<tr>
<td>setup_method()</td>
<td>(hyperSpy.tests.samfire.test_histogram_segmenter.TestHistogramSegmenter method)</td>
<td>443</td>
<td></td>
</tr>
</tbody>
</table>
setup_method() (hyperSpy.tests.samfire.test_red_chisq_weight.Test_Red_chisq_weight method), 443
setup_method() (hyperSpy.tests.samfire.test_global_strategy.TestGlobalStrategy method), 444
setup_method() (hyperSpy.tests.samfire.test_strategy.TestLocalSimple method), 444
setup_method() (hyperSpy.tests.samfire.test_strategy.TestLocalWithModel method), 444
setup_method() (hyperSpy.tests.samfire.test_strategy_list.TestStrategyList method), 444
setup_method() (hyperSpy.tests.samfire.test_utils.TestSamfireUtils method), 445
setup_method() (hyperSpy.tests.signal.test_assign_subclass.TestConvertBaseSignal method), 445
setup_method() (hyperSpy.tests.signal.test_assign_subclass.TestConvertComplexSignal method), 446
setup_method() (hyperSpy.tests.signal.test_assign_subclass.TestConvertComplexSignal1D method), 446
setup_method() (hyperSpy.tests.signal.test_assign_subclass.TestConvertSignal1D method), 446
setup_method() (hyperSpy.tests.signal.test_binned.TestModelBinned method), 446
setup_method() (hyperSpy.tests.signal.test_eds_sem.TestEnergyUnits method), 447
setup_method() (hyperSpy.tests.signal.test_eds_sem.TestGetLinesIntensinty method), 447
setup_method() (hyperSpy.tests.signal.test_eds_sem.TestMetadata method), 448
setup_method() (hyperSpy.tests.signal.test_eds_sem.TestToolsBulk method), 448
setup_method() (hyperSpy.tests.signal.test_eds_markers.TestEdsMarkers method), 448
setup_method() (hyperSpy.tests.signal.test_eds_tem.TestEdsTem method), 449
setup_method() (hyperSpy.samfire_utils.samfire_worker.Worker method), 450
setup_method() (hyperSpy.tests.samfire.test_red_chisq_weight.Test_Red_chisq_weight method), 449
setup_method() (hyperSpy.tests.samfire.test_red_chisq_weight.Test_Red_chisq_weight method), 450
setup_method() (hyperSpy.tests.signal.test_eds_tem.TestVacuumMask method), 450
setup_method() (hyperSpy.tests.signal.test_find_peaks1DoHaver.TestFindPeaks1D0haver method), 451
setup_method() (hyperSpy.tests.signal.test_folding.TestSignalFolding method), 451
setup_method() (hyperSpy.tests.signal.test_image.Test2D method), 452
setup_method() (hyperSpy.tests.signal.test_image.Test3D method), 452
setup_method() (hyperSpy.tests.signal.test_image.Test4D method), 452
setup_method() (hyperSpy.tests.signal.test_spectrum.Test2D method), 453
setup_method() (hyperSpy.tests.signal.test_spectrum.Test3D method), 453
setup_method() (hyperSpy.tests.signal.test_spectrum.Test4D method), 453
setup_method() (hyperSpy.samfire_utils.samfire_worker.Worker method), 454
setup_method() (hyperSpy.tests.signal.test_stack.TestUtilsStack method), 454
setup_method() (hyperSpy.tests.utils.test_material.TestWeightToFromAtomic method), 455
setup_method() (hyperSpy.samfire_utils.samfire_worker.Worker method), 456
setup_method() (hyperSpy.samfire_utils.samfire_worker.Worker method), 428
SFS_reader (class in hyperspy.io_plugins.bruker), 341
SFSTreeItem (class in hyperspy.io_plugins.bruker), 340
shape (hyperspy.external.tifffile.TiffPage attribute), 331
shape (hyperspy.external.tifffile.TiffSequence attribute), 329
shape () (hyperspy.axes.AxesManager property), 464
shape () (hyperspy.io_plugins.digital_micrograph.ImageObject attribute), 330
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class/Function</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>snap_position()</td>
<td>hyperspy.drawing.widget.DraggableWidgetBase property</td>
<td>296</td>
</tr>
<tr>
<td>snap_size()</td>
<td>hyperspy.drawing.widget.ResizableDraggableWidgetBase property</td>
<td>296</td>
</tr>
<tr>
<td>sobel_filter()</td>
<td>in module hyperspy.signals.signal1d</td>
<td>268</td>
</tr>
<tr>
<td>software()</td>
<td>hyperspy.external.tifffile.TiffPage attribute</td>
<td>335</td>
</tr>
<tr>
<td>someweight()</td>
<td>(class in hyperspy.tests.samfire.test_strategy)</td>
<td>444</td>
</tr>
<tr>
<td>span_selector_changed()</td>
<td>hyperspy.signal_tools.BackgroundRemoval method</td>
<td>535</td>
</tr>
<tr>
<td>span_selector_changed()</td>
<td>hyperspy.signal_tools.SpikesRemoval method</td>
<td>539</td>
</tr>
<tr>
<td>span_selector_switch()</td>
<td>hyperspy.signal_tools.ImageContrastEditor method</td>
<td>555</td>
</tr>
<tr>
<td>span_selector_switch()</td>
<td>hyperspy.signal_tools.SpanSelectorInSignal1D method</td>
<td>538</td>
</tr>
<tr>
<td>SpanROI</td>
<td>(class in hyperspy.roi)</td>
<td>497</td>
</tr>
<tr>
<td>SpanSelectorInSignal1D</td>
<td>(class in hyperspy.signal_tools)</td>
<td>538</td>
</tr>
<tr>
<td>spc_reader()</td>
<td>in module hyperspy.io_plugins.edax</td>
<td>351</td>
</tr>
<tr>
<td>spd_reader()</td>
<td>in module hyperspy.io_plugins.edax</td>
<td>352</td>
</tr>
<tr>
<td>SpecialSlicers</td>
<td>(class in hyperspy.misc.slicing)</td>
<td>396</td>
</tr>
<tr>
<td>SpecialSlicersSignal</td>
<td>(class in hyperspy.signal)</td>
<td>534</td>
</tr>
<tr>
<td>spectrum()</td>
<td>(hyperspy.models.edmodel.EDSMModel property)</td>
<td>410</td>
</tr>
<tr>
<td>spikes_removal_tool()</td>
<td>(hyperspy.signals.signal1d.Signal1D method)</td>
<td>262</td>
</tr>
<tr>
<td>SpikesRemoval</td>
<td>(class in hyperspy.signal_tools)</td>
<td>338</td>
</tr>
<tr>
<td>spin_orbit_branching_ratio</td>
<td>(hyperspy.components.voigt.Voigt attribute)</td>
<td>211</td>
</tr>
<tr>
<td>spin_orbit_splitting</td>
<td>(hyperspy.components.voigt.Voigt attribute)</td>
<td>211</td>
</tr>
<tr>
<td>spin_orbit_splitting_energy</td>
<td>(hyperspy.components.voigt.Voigt attribute)</td>
<td>211</td>
</tr>
<tr>
<td>split()</td>
<td>(hyperspy.signal.BaseSignal method)</td>
<td>521</td>
</tr>
<tr>
<td>spx_reader()</td>
<td>in module hyperspy.io_plugins.bruker</td>
<td>343</td>
</tr>
<tr>
<td>SquareWidget</td>
<td>(class in hyperspy.drawing.widgets.rectangles)</td>
<td>282</td>
</tr>
<tr>
<td>squeeze()</td>
<td>(hyperspy.signal.BaseSignal method)</td>
<td>522</td>
</tr>
<tr>
<td>squeeze_axes()</td>
<td>(in module hyperspy.signal_base)</td>
<td>282</td>
</tr>
<tr>
<td>stack()</td>
<td>(in module hyperspy.misc.utils)</td>
<td>404</td>
</tr>
<tr>
<td>start()</td>
<td>(hyperspy.samfire.Samfire method)</td>
<td>498, 500</td>
</tr>
<tr>
<td>start_listening()</td>
<td>(hyperspy.samfire_utils.samfire_worker.Worker method)</td>
<td>428</td>
</tr>
<tr>
<td>stash_active_state()</td>
<td>(in module hyperspy.misc.utils)</td>
<td>405</td>
</tr>
<tr>
<td>statistics()</td>
<td>(hyperspy.signals.hologram_image.HologramImage method)</td>
<td>247</td>
</tr>
<tr>
<td>std()</td>
<td>(hyperspy.signal.BaseSignal method)</td>
<td>522</td>
</tr>
<tr>
<td>stk_metadata</td>
<td>(hyperspy.external.tifffile.TiffFile attribute)</td>
<td>326</td>
</tr>
<tr>
<td>stop()</td>
<td>(hyperspy.samfire.Samfire method)</td>
<td>498</td>
</tr>
<tr>
<td>stop()</td>
<td>(hyperspy.samfire_utils.samfire_pool.SamfirePool method)</td>
<td>426, 427</td>
</tr>
<tr>
<td>stop_listening()</td>
<td>(hyperspy.samfire_utils.samfire_worker.Worker method)</td>
<td>428</td>
</tr>
<tr>
<td>store()</td>
<td>(hyperspy.model.BaseModel method)</td>
<td>491</td>
</tr>
<tr>
<td>store()</td>
<td>(hyperspy.signal.BaseSetMetadataItems method)</td>
<td>501</td>
</tr>
<tr>
<td>store()</td>
<td>(hyperspy.signal.ModelManager method)</td>
<td>533</td>
</tr>
<tr>
<td>store_current_parameters_in_map()</td>
<td>(hyperspy.component.Component method)</td>
<td>468</td>
</tr>
<tr>
<td>store_current_value_in_array()</td>
<td>(hyperspy.component.Parameter method)</td>
<td>471</td>
</tr>
<tr>
<td>store_current_values()</td>
<td>(hyperspy.model.BaseModel method)</td>
<td>482, 492</td>
</tr>
<tr>
<td>store_current_values()</td>
<td>(hyperspy.models.model1d.Model1D method)</td>
<td>418</td>
</tr>
<tr>
<td>str2bytes()</td>
<td>(in module hyperspy.external.tifffile)</td>
<td>338</td>
</tr>
<tr>
<td>str2num()</td>
<td>(in module hyperspy.misc.utils)</td>
<td>405</td>
</tr>
<tr>
<td>strategies()</td>
<td>(hyperspy.samfire.Samfire attribute)</td>
<td>498</td>
</tr>
<tr>
<td>StrategyList</td>
<td>(class in hyperspy.samfire)</td>
<td>500</td>
</tr>
<tr>
<td>stream_to_array()</td>
<td>(hyperspy.io_plugins.emd.FeiSpectrumStream method)</td>
<td>354</td>
</tr>
<tr>
<td>stream_to_array()</td>
<td>(in module hyperspy.misc.io.fei_stream_readers)</td>
<td>383</td>
</tr>
<tr>
<td>stream_to_sparse_array()</td>
<td>(hyperspy.io_plugins.emd.FeiSpectrumStream method)</td>
<td>354</td>
</tr>
<tr>
<td>stream_to_sparse_COO_array()</td>
<td>(in module hyperspy.misc.io.fei_stream_readers)</td>
<td>383</td>
</tr>
<tr>
<td>stripnull()</td>
<td>(in module hyperspy.external.tifffile)</td>
<td>337</td>
</tr>
<tr>
<td>strlist2enumeration()</td>
<td>(in module hyperspy.misc.utils)</td>
<td>405</td>
</tr>
</tbody>
</table>
test_datetime_to_serial_date() (in module hyperspy.tests.misc.test_date_time_tools), 438

test_default() (in module hyperspy.tests.mva.test_onmf), 442

test_default_init() (hyper spy.tests.samfire.test_strategy.TestLocalSimple method), 444

test_default_param() (hyper spy.tests.signal.test_eds_sem.Test_metadata method), 448

test_default_param() (hyper spy.tests.signal.test_eds_tem.Test_metadata method), 449

test_density_of_mixture() (in module hyperspy.tests.utils.test_material), 454

test_dielectric_function_binned_default() (test_export_free() (hyper spy.tests.component.test_component.TestGeneralMethods method), 432

test_disable_pixel_switching() (hyper spy.tests.component.test_component.TestMultidimensionalActive method), 432

test_disable_pixel_switching_current_off() (hyper spy.tests.component.test_component.TestMultidimensionalActive method), 432

test_disable_pixel_switching_current_on() (hyper spy.tests.component.test_component.TestMultidimensionalActive method), 432

test_dof_with_fit() (hyper spy.tests.model.test_chi_squared.TestChiSquared method), 440

test_dof_with_inactive_components() (hyper spy.tests.model.test_chi_squared.TestChiSquared method), 440

test_dof_with_p0() (hyper spy.tests.model.test_chi_squared.TestChiSquared method), 440

Test_eds_markers (class in hyperspy.tests.signal.test_eds_sem), 448

test_eds_sem_binned_default() (in module hyperspy.tests.signal.test_binned), 446

test_eds_tem_binned_default() (in module hyperspy.tests.signal.test_binned), 446

test_edx_cross_section_to_zeta() (hyper spy.tests.signal.test_eds_tem.Test_quantification method), 449

test_eels_spectrum_binned_default() (in module hyperspy.tests.signal.test_binned), 447

test_electron_range() (hyper spy.tests.signal.test_eds_sem.Test_tools_bulk method), 448

test_enable_pixel_switching_current_off() (test_function() (hyper spy.tests.component.test_component.TestMultidimensionalActive method), 432

test_enable_pixel_switching_current_on() (test_function() (hyper spy.tests.component.test_component.TestMultidimensionalActive method), 432

Index
spy.tests.component.test_exponential, 434

spy.tests.dictionary_tree_browser.TestDictionaryBrowser method, 455

test_hotkeys_in_six_dimensions() (hyperspy.tests.axes.test_axes_manager.TestAxesHotkeys method), 431

test_html_output() (hyperspy.tests.utils.test_print_known_signal_types.TestPrintKnownSignalTypes method), 454

test_ignore_full_message() (in module hyperspy.tests.misc.test_test_utils), 439

test_ignore_message_fails() (in module hyperspy.tests.misc.test_test_utils), 439

test_ignore_partial_message() (in module hyperspy.tests.misc.test_test_utils), 439

test_ignore_regex_message() (in module hyperspy.tests.misc.test_test_utils), 439

test_ignore_type() (in module hyperspy.tests.misc.test_test_utils), 439

test_ignore_type_fails() (in module hyperspy.tests.misc.test_test_utils), 439

Index 593
test_lazy_add_elements() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 448
test_lazy_add_lines() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 448
test_lazy_add_lines_auto() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 448
test_lazy_add_phase_ramp() (in module hyperspy.tests.signal.test_complex_signal2d), 447
test_lazy_are_microscope_parameters_missing() (hyperspy.tests.signal.test_eds_tem.Test_metadata method), 449
test_lazy_background_subtraction() (hyper-spy.tests.signal.test_eds_sem.Test_get_lines_intensity method), 447
test_lazy_beam_energy() (hyper-spy.tests.signal.test_eds_sem.Test_energy_units method), 447
test_lazy_chisq() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_chisq_in_range() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_chisq_with_fit() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_chisq_with_inactive_components() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_dof_with_fit() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_dof_with_inactive_components() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 449
test_lazy_default_param() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 449
test_lazy_dof_with_p0() (hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440
test_lazy_edx_cross_section_to_zeta() (hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 449
test_lazy_electron_range() (hyper-spy.tests.signal.test_eds_sem.Test_tools_bulk method), 448
test_lazy_eV() (hyper-spy.tests.signal.test_eds_sem.Test_get_lines_intentsity method), 447
test_lazy_eV() (hyper-spy.tests.signal.test_eds_sem.Test_get_lines_intentsity method), 447
test_lazy_find_peaks1D_oahaver_high_amp_thres() (hyper-spy.tests.signal.test_find_peaks1D_oahaver.TestFindPeaks1D method), 451
test_lazy_find_peaks1D_oahaver_zero_value_bug() (hyper-spy.tests.signal.test_find_peaks1D_oahaver.TestFindPeaks1D method), 451
test_lazy_get_calibration_from() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 448
test_lazy_get_calibration_from() (hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 449
test_lazy_intensity() (hyper-spy.tests.signal.test_eds_tem.Test_simple_model method), 450
test_lazy_intensity() (hyper-spy.tests.signal.test_eds_tem.Test_simple_model method), 450
test_lazy_line_energy() (hyper-spy.tests.signal.test_eds_sem.Test_energy_units method), 447
test_lazy_metadata() (hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 449
test_lazy_model_class() (hyper-spy.tests.model.test_fancy_indexing.TestModelIndexingClass method), 441
test_lazy_model_navigation_indexer_slice() (hyper-spy.tests.signal.test_fancy_indexing.TestModelIndexing method), 440
test_lazy_model_signal_indexer_slice() (hyper-spy.tests.signal.test_fancy_indexing.TestModelIndexing method), 440
test_lazy_quant_cross_section() (hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 449
test_lazy_quant_cross_section_units() (hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 449
test_lazy_quant_element_order() (hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450
spy.tests.signal.test eds_tem.Test_quantification (method), 450

spy.tests.signal.test eds_tem.TestLazyQuantification (method), 449

spy.tests.signal.test eds_tem.TestLazyToEELSAndBack (method), 446

spy.tests.signal.test eds getTimeQuantification (method), 450

spy.tests.signal.test eds getTimeQuantification (method), 450

spy.tests.signal.test eds_sem.TestLazyToImage (method), 452

spy.tests.signal.test eds_sem.TestLazySEMToTEM (method), 448

spy.tests.signal.test eds_sem.TestLazyToEELSAndBack (method), 452

spy.tests.signal.test eds_sem.TestLazyToSignal2D (method), 452

spy.tests.signal.test eds_sem.TestLazyUnfoldedNavigationByKeyword (method), 451

spy.tests.signal.test eds_sem.TestLazyUnfoldedNavigationByKeyword (method), 451

Index 595
test_lazy_unfolded_nothing_by_keyword() (hyperspy.signals.test_folding.TestSignalFolding method), 451

test_lazy_unfolded_repr() (hyperspy.signals.test_folding.TestSignalFolding method), 451

test_lazy_unfolded_signal_by_keyword() (hyperspy.signals.test_folding.TestSignalFolding method), 451

test_lazy_vacuum_mask() (hyperspy.signals.test_eds_tem.Test_vacum_mask method), 450

test_lazy_with_signals_examples() (hyperspy.signals.test_eds_tem.Test_get_lines_intentisity method), 447

test_line_energy() (hyperspy.signals.test_eds_sem.Test_get_lines_intentisity method), 447

test_linfit() (in module hyperspy.external.mpfit.test_mpfit), 313

test_load_stackbuilder_imagestack() (hyperspy.tests.io.test_dm_stackbuilder_plugin.TestStackBuilder method), 436

test_low_high_value() (hyperspy.tests.axes.test_axes_manager.TestAxesManagerScaleOffset method), 431

test_mac() (in module hyperspy.tests.utils.test_material), 454

test_manual_add_line() (hyperspy.tests.signal.test_eds_tem.Test_eds_markers method), 449

test_manual_remove_element() (hyperspy.tests.signal.test_eds_tem.Test_eds_markers method), 449

test_map() (hyperpy.tests.samfire.test_goodness_of_fit.TestInformationCriteria method), 443

test_map() (hyperpy.tests.samfire.test_goodness_of_fit.TestRedChisq method), 443

test_map_noslice() (hyperpy.tests.samfire.test_red_chisq_weight.Test_Red_chisq_weight method), 445

Test_metadata (class in hyperspy.signals.test_eds_sem), 447

Test_metadata (class in hyperspy.signals.test_eds_tem), 449

Test_metadata() (hyperpy.tests.io.test_msa.TestExample1 method), 437

Test_metadata() (hyperpy.tests.io.test_msa.TestExample1WrongDate method), 437

Test_metadata() (hyperpy.tests.io.test_msa.TestExample2 method), 437

Test_metadata() (hyperpy.tests.signal.test_eds_tem.Test_quantification method), 450

Test_minimum_metadata_example() (in module hyperspy.tests.io.test_msa), 437

Test_mixture_mac() (in module hyperspy.tests.model.test_fancy_indexing.TestModelIndexingClass method), 441

Test_model_class() (hyperpy.tests.model.test_fancy_indexing.TestModelIndexingClass method), 441

Test_model_navigation_indexer_slice() (hyperpy.tests.model.test_fancy_indexing.TestModelIndexingClass method), 441

Test_nearest_indices_cent1() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_cent2() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_cent3() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_cent4() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_ind1() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_ind2() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_ind3() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445

Test_nearest_indices_ind4() (hyperpy.tests.samfire.test_weight_fromatomic.TestNearestIndices method), 445
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_nearest_indices_ind5()</td>
<td>(hyper-spy.tests.samfire.test_utils.TestSamfireUtils method), 445</td>
</tr>
<tr>
<td>test_no_function_nd()</td>
<td>(hyper-spy.tests.component.test_components2D.TestExpression2D spy.tests.signal.test_eds_tem method), 449</td>
</tr>
<tr>
<td>test_no_function_nd_signal()</td>
<td>(hyper-spy.tests.component.test_components2D.TestExpression2D method), 433</td>
</tr>
<tr>
<td>test_no_rotation()</td>
<td>(hyper-spy.tests.component.test_components2D.TestExpression2D method), 433</td>
</tr>
<tr>
<td>test_offset_after_rebin()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_metadata method), 449</td>
</tr>
<tr>
<td>test_package_values()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestGlobalStrategy method), 444</td>
</tr>
<tr>
<td>test_parameters()</td>
<td>(hyper-spy.tests.io.test_msa.TestExample1 method), 437</td>
</tr>
<tr>
<td>test_plot_auto_add()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_eds_markers method), 439</td>
</tr>
<tr>
<td>test_plotting_active_component_binned()</td>
<td>(hyper-spy.tests.component.test_component.TestCallMethods method), 444</td>
</tr>
<tr>
<td>test_plotting_active_component_notbinned()</td>
<td>(hyper-spy.tests.component.test_component.TestCallMethods method), 444</td>
</tr>
<tr>
<td>test_plotting_active_component_out_of_range()</td>
<td>(hyper-spy.tests.component.test_component.TestCallMethods method), 444</td>
</tr>
<tr>
<td>test_plotting_not_active_component()</td>
<td>(hyper-spy.tests.component.test_component.TestCallMethods method), 444</td>
</tr>
<tr>
<td>test_quant_cross_section()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_quant_cross_section_units()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_quant_element_order()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_quant_lorimer()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_quant_zeros()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_quant_zeta()</td>
<td>(hyper-spy.tests.signal.test_eds_tem.Test_quantification method), 450</td>
</tr>
<tr>
<td>test_red_chisq()</td>
<td>(hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440</td>
</tr>
<tr>
<td>test_red_chisq_with_fit()</td>
<td>(hyper-spy.tests.model.test_chi_squared.TestChiSquared method), 440</td>
</tr>
<tr>
<td>test_refresh_nooverwrite()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestLocalSimple method), 444</td>
</tr>
<tr>
<td>test_refresh_nooverwrite_given()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestLocalSimple method), 444</td>
</tr>
<tr>
<td>test_refresh_overwrite()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestLocalSimple method), 444</td>
</tr>
<tr>
<td>test_refresh_overwrite_given()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestLocalSimple method), 444</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_c()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_c_slices()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_f()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_cordermask_from_cmasked()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_cordermask_from_cmasked_slices()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_refresh_nooverwrite_nogiven()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestGlobalStrategy method), 444</td>
</tr>
<tr>
<td>test_refresh_overwrite_nogiven()</td>
<td>(hyper-spy.tests.samfire.test_strategy.TestGlobalStrategy method), 444</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_c()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_c_slices()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_corder_from_f()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_cordermask_from_cmasked()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
<tr>
<td>test_regular_array2rgbx_cordermask_from_cmasked_slices()</td>
<td>(hyper-spy.tests.misc.test_rgbtools.TestRGBTools method), 438</td>
</tr>
</tbody>
</table>
...
update_range() (hyper spy.drawing._widgets.range.ModifiableSpanSelector method), 280
update_span_selector_traits() (hyper spy.signal_tools.ImageContrastEditor method), 535
update_span_selector_traits() (hyper spy.signal_tools.SpanSelectorInSignal1D method), 538
update_spectrum_line() (hyper spy.signal_tools.SpikesRemoval method), 539
user (hyperspy.io_plugins.emd.EMD attribute), 352
vacuum_mask() (hyper spy.signals.eds_tem.EDSTEM_mixin method), 235
validate() (hyperspy.component.NoneFloat method), 468
validate_input() (hyper spy.external.astroML.bayesian_blocks.FitnessFunc method), 299
validate_input() (hyper spy.external.astroML.bayesian_blocks.RegularEvents method), 299
value (hyperspy.axes.AxesManager attribute), 464
value (hyperspy.component.Parameter attribute), 468
value (hyperspy.external.tifffile.TiffTag attribute), 336
value2index() (hyperspy.axes.DataAxis method), 465
value_range_to_indices() (hyperspy.axes.DataAxis method), 465
valuemax() (hyperspy.signals.lazy.LazySignal method), 253
valuemax() (hyperspy.signal.BaseSignal method), 525
valuemin() (hyperspy.signals.lazy.LazySignal method), 253
valuemin() (hyperspy.signal.BaseSignal method), 526
valueoffset (hyperspy.external.tifffile.TiffTag attribute), 336
values() (hyperspy.samfire_utils.strategy.GlobalStrategy method), 429
values() (hyperspy.samfire_utils.strategy.LocalStrategy method), 429
var() (hyperspy.signal.BaseSignal method), 526
variance() (hyperspy_components.skew_normal.SkewNormal method), 211
VerticalLine (class in hyperspy.drawing_markers.vertical_line), 277
VerticalLineSegment (class in hyperspy.drawing_markers.vertical_line_segment), 277
VerticalLineWidget (class in hyperspy.drawing_widgets.vertical_line), 283
Vmax (class in hyperspy_components.eels_vignetting), 197
VisibleDeprecationWarning, 477
vmax() (hyperspy.drawing.image.ImagePlot property), 284
vmin() (hyperspy.drawing.image.ImagePlot property), 284
Voigt (class in hyperspy_components.voigt), 211
Voigt() (in module hyperspy_components.voigt), 212
VolumePlasmonDrude (class in hyperspy_components.volume_plasmon_drude), 213
W
warnsA() (in module hyperspy_tests.misc.test_test_utils), 439
warnsB() (in module hyperspy_tests.misc.test_test_utils), 439
warnsC() (in module hyperspy_tests.misc.test_test_utils), 439
weight_to_atomic() (in module hyperspy_misc.material), 393
Widget1DBase (class in hyperspy.drawing.widget), 296
Widget2DBase (class in hyperspy.drawing.widget), 296
WidgetBase (class in hyperspy.drawing.widget), 297
width() (hyperspy.drawing_widgets.rectangles.RectangleWidget property), 282
width() (hyperspy.roi.RectangularROI property), 497
Worker (class in hyperspy.samfire_utils.samfire_worker), 428
workers (hyperspy.samfire.Samfire attribute), 498
write() (hyperspy.external.tifffile.FileHandle method), 331
write_array() (hyperspy.external.tifffile.FileHandle method), 331
write_empty() (hyperspy.external.tifffile.FileHandle method), 331
write_raw() (in module hyperspy.io_plugins.ripple), 359
write_rpl() (in module hyperspy.io_plugins.ripple), 359
write_signal() (in module hyperspy_io_plugins.hspy), 356
WrongObjectError, 477
x() (hyperspy.roi.RectangularROI property), 497
xml2dict() (in module hyperspy_external.tifffile), 338
X
xray_lines_model() (in module hyperspy.misc.eds.utils), 375
xray_range() (in module hyperspy.misc.eds.utils), 375

Y

y() (hyperspy.roi.RectangularROI property), 497

Z

zeta_to_edx_cross_section() (in module hyperspy.misc.eds.utils), 376