# CONTENTS

## 1 HyperSpy User Guide
1.1 Introduction .................................................. 1
1.2 What’s new .................................................. 2
1.3 Changelog .................................................... 6
1.4 Installing HyperSpy .......................................... 20
1.5 Getting started ................................................ 23
1.6 The Signal class .............................................. 33
1.7 Interactive Operations and Region of Interest (ROI) .... 61
1.8 Signal1D Tools ............................................... 69
1.9 Signal2D Tools ............................................... 73
1.10 Data visualization ........................................... 78
1.11 Machine learning ........................................... 109
1.12 Model fitting ................................................ 135
1.13 Electron Energy Loss Spectroscopy ....................... 158
1.14 Energy-Dispersive X-ray Spectrometry (EDS) ......... 164
1.15 Dielectric function tools ................................... 180
1.16 Electron Holography ....................................... 181
1.17 Loading and saving data ................................... 185
1.18 Events ......................................................... 206
1.19 Working with big data ...................................... 208
1.20 Metadata structure .......................................... 213
1.21 Bibliography ................................................ 220

## 2 HyperSpy Developer Guide
2.1 Introduction .................................................. 223
2.2 Using Git and GitHub ......................................... 224
2.3 Running and writing tests .................................... 226
2.4 Writing documentation ....................................... 229
2.5 Coding style .................................................. 230
2.6 Tips for writing methods that work on lazy signals ... 230
2.7 Speeding up code ............................................. 231
2.8 Writing packages that extend HyperSpy .................. 232
2.9 Useful information ........................................... 236

## 3 hyperspy
3.1 hyperspy package ............................................ 237

## 4 Credits

## 5 Citing HyperSpy
6  Indices and tables

Bibliography

Python Module Index

Index
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

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

1.2.1 Current Version

v1.6

NEW

- Support for the following file formats:
  - SUR and PRO format
  - Phenom ELID format
  - Nexus
  - USID
  - EMPAD format
  - Prismatic EMD format, see EMD
- `print_edges_near_energy()` method that, if the hyperspy-gui-ipywidgets package is installed, includes an awesome interactive mode. See Elemental composition of the sample.
- Model asymmetric line shape components:
  - Doniach
  - SplitVoigt
- EDS absorption correction.
- Argand diagram for complex signals.
- Multiple peak finding algorithms for 2D signals.
- Cluster analysis.

Enhancements

- The `get_histogram()` now uses numpy’s `np.histogram_bin_edges()` and supports all of its `bins` keyword values.
- Further improvements to the contrast adjustment tool. Test it by pressing the `h` key on any image.
- The following components have been rewritten using `hyperspy._components.expression.Expression`, boosting their speeds among other benefits.
  - `hyperspy._components.arctan.Arctan`
  - `hyperspy._components.voigt.Voigt`
  - `hyperspy._components.heaviside.HeavisideStep`
- The model fitting `fit()` and `multifit()` methods have been vastly improved. See Fitting the model to the data and the API changes section below.
• New serpentine iteration path for multi-dimensional fitting. See *Fitting multidimensional datasets.*

• The `plot_spectra()` function now listens to events to update the figure automatically. See *this example.*

• Improve thread-based parallelism. Add `max_workers` argument to the `map()` method, such that the user can directly control how many threads they launch.

• Many improvements to the `decomposition()` and `blind_source_separation()` methods, including support for scikit-learn like algorithms, better API and much improved documentation. See *Machine learning* and the API changes section below.

• Add option to calculate the absolute thickness to the EELS `estimate_thickness()` method. See *Thickness estimation.*

• Vastly improved performance and memory footprint of the `estimate_shift2D()` method.

• The `remove_background()` method can now remove Doniach, exponential, Lorentzian, skew normal, split Voigt and Voigt functions. Furthermore, it can return the background model that includes an estimation of the reduced chi-squared.

• The performance of the maximum-likelihood PCA method was greatly improved.

• All ROIs now have a `__getitem__` method, enabling e.g. using them with the unpack `*` operator. See *Slicing using ROIs* for an example.

• New syntax to set the contrast when plotting images. In particular, the `vmin` and `vmax` keywords now take values like `vmin="30th"` to clip the minimum value to the 30th percentile. See *Fast Fourier Transform (FFT)* for an example.

• The `plot()` and `plot()` methods take a new keyword argument `autoscale`. See *Customising image plot* for details.

• The contrast editor and the decomposition methods can now operate on complex signals.

• The default colormap can now be set in *preferences.*

**API changes**

• The `plot()` keyword argument `saturated_pixels` is deprecated. Please use `vmin` and/or `vmax` instead.

• The `load()` keyword argument `dataset_name` has been renamed to `dataset_path`.

• The `set_signal_type()` method no longer takes `None`. Use the empty string "" instead.

• The `get_histogram()` bins keyword values have been renamed as follows for consistency with numpy:
  - "scotts" -> "scott",
  - "freedman" -> "fd"

• Multiple changes to the syntax of the `fit()` and `multifit()` methods:

• The `fitter` keyword has been renamed to `optimizer`.

• The values that the `optimizer` keyword take have been renamed for consistency with scipy:
  - "fmin" -> "Nelder-Mead",
  - "fmin_cg" -> "CG",
  - "fmin_ncg" -> "Newton-CG",
  - "fmin_bfgs" -> "BFGS",
  - "fmin_l_bfgs_b" -> "L-BFGS-B".
- "fmin_tnc" -> "TNC",
- "fmin_powell" -> "Powell",
- "mpfit" -> "lm" (in combination with "bounded=True"),
- "leastsq" -> "lm",

• Passing integer arguments to parallel to select the number of workers is now deprecated. Use parallel=True, max_workers={value} instead.
• The method keyword has been renamed to loss_function.
• The loss_function value "ml" has been renamed to "ML-poisson".
• The grad keyword no longer takes boolean values. It takes the following values instead: "fd", "analytical", callable or None.
• The ext_bounding keyword has been deprecated and will be removed. Use bounded=True instead.
• The min_function keyword argument has been deprecated and will be removed. Use loss_function instead.
• The min_function_grad keyword arguments has been deprecated and will be removed. Use grad instead.
• The iterpath default will change from 'flyback' to 'serpentine' in HyperSpy version 2.0.
• The following BaseModel methods are now private:
  - set_boundaries()
  - set_mpfit_parameters_info()
  - set_boundaries()

• The comp_label keyword of the machine learning plotting functions has been renamed to title.
• The orpca constructor's learning_rate keyword has been renamed to subspace_learning_rate
• The orpca constructor's momentum keyword has been renamed to subspace_momentum
• The svd_pca constructor's centre keyword values have been renamed as follows:
  - "trials" -> "navigation"
  - "variables" -> "signal"
• The bounds keyword argument of the decomposition() is deprecated and will be removed.
• Several syntax changes in the decomposition() method:
  - Several algorithm keyword values have been renamed as follows:
    * "svd": "SVD",
    * "fast_svd": "SVD",
    * "nrm": "NRM",
    * "fast_mlpca": "MLPCA",
    * "mlpca": "MLPCA",
    * "RPCA_GoDec": "RPCA",
  - The polyfit argument has been deprecated and will be removed. Use var_func instead.
1.2.2 Previous Versions

v1.5.2

This is a maintenance release that adds compatibility with Numpy 1.17 and Dask 2.3.0 and fixes a bug in the Bruker reader. See the issue tracker for details.

v1.5.1

This is a maintenance release that fixes some regressions introduced in v1.5. Follow the following links for details on all the bugs fixed.

v1.5

NEW

- 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 \texttt{hyperspy._components.polynomial_deprecated.Polynomial} component will be deprecated in HyperSpy 2.0 in favour of the new \texttt{hyperspy._components.polynomial.Polynomial} component, that is based on \texttt{hyperspy._components.expression.Expression} and has an improved API. To start using the new component pass the \texttt{legacy=False} keyword to the \texttt{hyperspy._components.polynomial_deprecated.Polynomial} component constructor.

\textbf{For developers}

• Drop support for python 3.5

• New extension mechanism that enables external packages to register HyperSpy objects. See \textit{Writing packages that extend HyperSpy} for details.

\section*{1.3 Changelog}

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.

This release fixes compatibility issues with Python 3.7.

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

• Support for three new file formats:
  – Reading FEI’s Velox EMD file format based on the HDF5 open standard. See \textit{EMD (Velox)}.
  – Reading Bruker’s SPX format. See \textit{SPX format}.
  – Reading and writing the mrcz open format. See \textit{MRCZ}.

• New \texttt{artificial_data} module which contains functions for generating artificial data, for use in things like docstrings or for people to test HyperSpy functionalities. See \textit{Loading example data and data from online databases}.

• New \texttt{fft()} and \texttt{ifft()} signal methods. See \textit{Fast Fourier Transform (FFT)}.

• New \texttt{statistics()} method to compute useful hologram parameters. See \textit{Further processing of complex wave and phase}.

• Automatic axes units conversion and better units handling using \texttt{pint}. See \textit{Using quantity and converting units}.

• New \texttt{Line2DROIangle()} method. See \textit{Region Of Interest (ROI)} for details.

• \texttt{plot_images()} improvements (see \textit{Plotting several images} for details):
  – The \texttt{cmap} option of \texttt{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 \textit{Data visualization}.

• The \texttt{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 \textit{Background removal} for details.
- The `plot()` method now takes a `norm` keyword that can be “linear”, “log”, “auto” or a matplotlib norm. See *Customising image plot* for details. Moreover, there are three new extra keyword arguments, `fft_shift` and `power_spectrum`, that are useful when plotting fourier transforms. See *Fast Fourier Transform (FFT)*.

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

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

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

- `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 `hyper-spy_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.

- 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 `hyper-spy_gui_ipywidgets` package. See *Starting Python in Windows*.

- The `vmin` and `vmax` arguments of the `plot_images()` function now accept lists to enable setting these parameters for each plot individually.

- The `plot_decomposition_results()` and `plot_bss_results()` methods now makes a better guess of the number of navigators (if any) required to visualise the components. (Previously they were always plotting four figures by default.)

- All functions that take a signal range can now take a `SpanROI`.

- The following ROIs can now be used for indexing or slicing (see *here* for details):
  - `Point1DROI`
  - `Point2DROI`
  - `SpanROI`
  - `RectangularROI`

- 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 split 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`
  - `Machine_learning.multiple_files`
  - `Machine_learning.same_window`
  - `Plot.default_style_to_compare_spectra`
  - `Plot.plot_on_load`
  - `Plot.pylab_inline`
  - `EELS.fine_structure_width`
  - `EELS.fine_structure_active`
  - `EELS.fine_structure_smoothing`
  - `EELS.synchronize_cl_with_ll`
  - `EELS.preedge_safe_window_width`
  - `EELS.min_distance_between_edges_for_fine_structure`

- New `Preferences.GUIs` section to enable/disable the installed GUI toolkits.

- In addition to adding ipywidgets GUI elements, the traitsui GUI elements have been splitted into a separate package. See the new `hyperspy_gui_traitsui` package.

- The new `ui_registry` enables easy connection of external GUI elements to HyperSpy. This is the mechanism used to split the traitsui and ipywidgets GUI elements.

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

- 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()`.
- Easier creation of `Expression` components using substitutions. See the `User Guide for details`.
- `Expression` takes two dimensional functions that can automatically include a rotation parameter. See the `User Guide for details`. 
• Better support for EMD files.
• The scree plot got a beauty treatment and some extra features. See *Scree plots*.
• `map()` can now take functions that return differently-shaped arrays or arbitrary objects, see *Iterating external functions with the map method*.
• Add support for stacking multi-signal files. See *Loading multiple files*.
• Markers can now be saved to hdf5 and creating many markers is easier and faster. See *Markers*.
• Add option to save to HDF5 file using the “.hspy” extension instead of “.hdf5”. See *HSpy - HyperSpy’s HDF5 Specification*. This will be the default extension in HyperSpy 1.3.
• Most of HyperSpy plotting features are now covered by unitests. See *Plot testing*.
• unitests migrated from nose to pytest. See *Running and writing tests*.

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

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

• 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.

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

• Transposing (changing signal spaces).
• Protochips log reader.
• `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.

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

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

• 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.
• Smart Adaptive Multi-dimensional Fitting (SAMFire).
• Store models in hdf5 files.
• Add fancy indexing to Model.
• Two-dimensional model fitting.
• Z-factors quantification.
• Cross section quantification.
• EDS curve fitting.
• X-ray absorption coefficient database.
• 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.
• 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.
• 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.
• 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.
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.

• Widgets to interact with the model components in the Jupyter Notebook. See here and #1007.

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.

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

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.

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.

- 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.
- 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.
- 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.
- HDF5 file format now supports saving lists, tuples, binary strings and signals in metadata.
- 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.
- 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 incorporates the vacuum mask
• *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`

This is a maintenance release. A list of fixed issues is available in the 0.7.3 milestone in the github repository.
This is a maintenance release. A list of fixed issues is available in the 0.7.2 milestone in the github repository.
This is a maintenance release. A list of fixed issues is available in the 0.7.1 milestone in the github repository.
• Add suspend/resume model plot updating. See *Visualizing the model*.
• 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.
• Improved support for FEI’s emi and ser files.
• Improved support for Gatan’s dm3 files.
• Add support for reading Gatan’s dm4 files.
• 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.
• 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.
• 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.
• 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 Clustering plots.
• New methods to plot the decomposition and blind source separation results that support n-dimensional loadings. See Cluster analysis.
• New `Signal` subclass, `DielectricFunction`.
• 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()`.
• New signal, `EDSSpectrum` specialized in EDS data analysis, with subsignal for EDS with SEM and with TEM: `EDSSEMSpectrum` 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.

1.3. Changelog
• 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 *Plotting*

• 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()`

• 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.

• 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)})

• 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.
• 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.
• EELSSPectrum.find_low_loss_centre was renamed to estimate_zero_loss_peak_centre.
• EELSSPectrum.calculate_FWHM was renamed to estimate_FWHM.
• 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 \textit{elements}.
• The ripple file format now supports storing the beam energy, the collection and the convergence angle.
• 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.
• \texttt{spatial\_mask} was renamed to \texttt{navigation\_mask}.
• \texttt{Signal1D} and \texttt{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.
• \texttt{\_slicing\_axes} was renamed to \texttt{signal\_axes}.
• \texttt{\_non\_slicing\_axes} to \texttt{navigation\_axes}.
• All the Model * \texttt{\_in\_pixels} methods were renamed to \texttt{\_\*\_in\_pixel}.
• \texttt{EELSCLEdge.fs\_state} was renamed to \texttt{fine\_structure\_active}.
• \texttt{EELSCLEdge.fs\_list} was renamed to \texttt{fine\_structure\_coeff}.
• \texttt{EELSCLEdge.fs\_emax} was renamed to \texttt{fine\_structure\_width}.
• \texttt{EELSCLEdge.freedelta} was renamed to \texttt{free\_energy\_shift}.
• \texttt{EELSCLEdge.delta} was renamed to \texttt{energy\_shift}.
• A value of True in a mask now means that the item is masked all over HyperSpy.
• 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 \texttt{\_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.
• Non-ascii characters breaking IO and print features fixed.
• Loading of multiple files at once using wildcards fixed.
• Remove broken hyperspy-gui script.
• Remove unmaintained and broken 2D peak finding and analysis features.
• In EELS automatic background feature creates a PowerLaw component, adds it to the model and add it to a variable in the user namespace. The variable has been renamed from bg to background.
• pes_gaussian Component renamed to pes_core_line_shape.
• 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.
• 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 calibrabrion UIs were freezing when the plots where closed before closing the UI window.
• to_spectrum was transposing the navigation dimension.
• variance2one was operating in the wrong axis.
• when closing the plots of a model, the UI object was not being destroyed.
• when plotting an image the title was not displayed.
• when the axis size was changed (e.g. after cropping) the set_signal_dimension method was not being called.
• when using transform the data was being centered and the resulting scores were wrong.
• in decomposition V rename to explained_variance.
• In FixedPattern, default interpolation changed to linear.
• Line and parabole 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 availabe in the interactive session.
• Several changes to the interface.
• The documentation was updated to reflex the last changes.
• The microscopes.csv file was updated so it no longer contains the Orsay VG parameters.
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 the *Installation in an Anaconda/Miniconda distribution* section.

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

**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](https://github.com/hyperspy/hyperspy-bundle)

1.4.2 Installation in an Anaconda/Miniconda distribution

Anaconda or Miniconda is recommended for the best performance (numpy is compiled using the Intel MKL libraries) and the easiest installation. HyperSpy is packaged in the *conda-forge* channel and can be installed easily using the `conda` package manager:

1. Download and install Anaconda if necessary. 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.

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

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

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

**Note:** Using `-c conda-forge` is only necessary when the conda-forge is not already added to the conda configuration, see the conda-forge documentation for more details.
Further information

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

- use Miniconda instead of Anaconda, if you are installing a python distribution from scratch: Miniconda installs very few packages so satisfying all dependencies is relatively simple.
- install hyperspy in a new environment. The following example illustrates creating a new environment named hspy_environment, activating it and installing hyperspy in the new environment.

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

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

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

To learn more about the Anaconda eco-system:

- Choose between Anaconda or Miniconda?
- Understanding conda and pip.

1.4.3 Installation using pip

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

To install all hyperspy functionalities, run:

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

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

```
$ pip install hyperspy
```

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

- `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,
- `mrcz` to install the mrcz plugin,
- `speed` install optional libraries that speed up some functionalities,
- `tests` to install required libraries to run HyperSpy’s unit tests,
• `build-doc` to install required libraries to build HyperSpy's documentation,
• `dev` to install all the above,
• `all` to install all the above except the development requirements (`tests`, `build-doc` and `dev`).

For example:

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

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

### 1.4.4 Install development version

#### Clone the hyperspy repository

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

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

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

#### Installation in a Anaconda/Miniconda distribution

Install the runtime and development dependencies requirements using conda:

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

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

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

```
$ pip install -e . --no-deps
```

#### Installation in other (non-system) Python distribution

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

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

All required dependencies are automatically installed by `pip`. If you don’t want to install all dependencies and only install some of the optional dependencies, use the corresponding selector as explained in the *Installation using pip* section.
Installation in a system Python distribution

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

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

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

Creating Debian/Ubuntu binaries

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

```
$ ./release_debian
```

**Warning:** For this to work, the following packages must be installed in your system Python

- python-stdeb
- debhelper
- dpkg-dev
- python-argparser

are required.

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:

```
$ 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.):

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

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

```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 if you are planning to perform interactive data analysis in the Jupyter Notebook. Otherwise, you can simply disable the warning in preferences as explained below.

- The `hyperspy-gui-traitsui` package is installed and you are using an incompatible matplotlib backend (e.g. `notebook`, `nbagg` or `widget`).
  - If you want to use the traitsui GUI, use the `qt` matplotlib backend instead.
  - Alternatively, if you prefer to use the `notebook` or `widget` matplotlib backend, and if you don’t want to see the (harmless) warning, make sure that you have the `hyperspy-gui-ipywidgets` installed and disable the traitsui GUI in the preferences.

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:
>>> 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.:

>>> hs?
>>> hs.load?
>>> hs.signals?

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

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:

>>> 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.:

>>> # 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.

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

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

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.
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, about reading the first lines of all pages, then the second and so on.

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

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

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

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

### 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>)>
```

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

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

![AxesManager ipywidgets GUI.

or the `DataAxis`, e.g.:

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

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

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

Alternatively, the “current position” can be changed programmatically by directly accessing `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 performing a fit over an entire spectrum image. The `indices` must be provided as a tuple, with the same length as the number of navigation dimensions:

```python
>>> s.axes_manager.indices = (5, 4)
```
Fig. 2: DataAxis ipywidgets GUI.

Fig. 3: Navigation sliders ipywidgets GUI.
1.5.11 Using quantity and converting units

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

```python
>>> 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'

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:

```python
>>> q = s.axes_manager[0].offset_as_quantity
>>> type(q)
<class '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`:

```python
>>> 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
    title = NbO2_Nb_M_David_Bach,_Wilfried_Sigle_217

>>> s.original_metadata
    DATATYPE = XY
    DATE =
    FORMAT = EMSA/MAS Spectral Data File
    NCOLUMNS = 1.0
    NPINTS = 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

>>> s.metadata.TEM.microscope = "STEM VG"
>>> s.metadata
    TEM
        EELS
            collection_angle = 20
            beam_energy = 100
            convergence_angle = 10
            microscope = STEM VG
            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
```
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:

![Preferences user interface](image)

Fig. 4: Preferences user interface.

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

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

```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
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)
```

1.6 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 =
```

1.6.3 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:
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).

<table>
<thead>
<tr>
<th><code>BaseSignal subclass</code></th>
<th><code>signal_dimension</code></th>
<th><code>signal_type</code></th>
<th><code>dtype</code></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()
```

<table>
<thead>
<tr>
<th>signal_type</th>
<th>aliases</th>
<th>class name</th>
<th>package</th>
</tr>
</thead>
<tbody>
<tr>
<td>DielectricFunction</td>
<td>dielectric function</td>
<td>DielectricFunction</td>
<td>hyperspy</td>
</tr>
<tr>
<td>EDS_SEM</td>
<td></td>
<td>EDSSEMSpectrum</td>
<td>hyperspy</td>
</tr>
<tr>
<td>EDS_TEM</td>
<td></td>
<td>EDSTEMSpectrum</td>
<td>hyperspy</td>
</tr>
<tr>
<td>EELS</td>
<td>TEM EELS</td>
<td>EELSSpectrum</td>
<td>hyperspy</td>
</tr>
<tr>
<td>hologram</td>
<td></td>
<td>HologramImage</td>
<td>hyperspy</td>
</tr>
<tr>
<td>MySignal</td>
<td></td>
<td>MySignal</td>
<td>hspy_ext</td>
</tr>
</tbody>
</table>

**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
```

(continues on next page)
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>
<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 these methods perform the operation over *all* navigation axes.

Example:

```python
>>> s = hs.signals.BaseSignal(np.random.random((2, 4, 6)))
>>> s.axes_manager[0].name = 'E'
>>> s
<BaseSignal, title: , dimensions: (|6, 4, 2)>
>>> # by default perform operation over all navigation axes
>>> s.sum()
<BaseSignal, title: , dimensions: (|6, 4, 2)>
>>> # can also pass axes individually
>>> s.sum('E')
<Signal2D, title: , dimensions: (|4, 2)>
>>> # or a tuple of axes to operate on, with duplicates, by index or directly
>>> ans = s.sum((-1, s.axes_manager[1], 'E', 0))
>>> ans
<BaseSignal, title: , dimensions: (|1)>
```

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)>
```  
```python
>>> np.exp(s)
<Signal1D, title: exp(A), dimensions: (|2)>
>>> np.exp(s).data
array([ 1. , 2.71828183])
```  
```python
>>> np.power(s, 2)
<Signal1D, title: power(A, 2), dimensions: (|2)>
>>> np.add(s, s)
<Signal1D, title: add(A, A), dimensions: (|2)>
```  
```python
>>> 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 unfucs) that operate on numpy arrays can also operate on `BaseSignal` instances, however they return a numpy array instead of a `BaseSignal` instance e.g.:

```python
>>> np.angle(s)
array([ 0., 0.])
```  
### 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. \([x, y, z, \ldots]\) (HyperSpy) vs \([\ldots, z, y, x]\) (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].data
array([0, 1, 2, 3, 4, 5])
>>> s.isig[5::2]
<Signal1D, title: , dimensions: (|3)>
>>> s.isig[5::2].data
array([5, 7, 9])
```

Unlike numpy, HyperSpy supports indexing using decimal numbers or 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])
>>> s.axes_manager[0].units = 'µm'
>>> s.isig[:'2000 nm'].data
array([0, 1, 2, 3])
```

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

```python
>>> s = hs.signals.Signal1D(np.arange(10))
>>> s
```
```python
>>> s = s.isig[::-2]
>>> si.data
array([0, 2, 4, 6, 8])
```

```python
>>> s.data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

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

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

```python
>>> s = hs.signals.Signal1D(np.arange(2*3*4).reshape((2,3,4)))
>>> s
<Signal1D, title: , dimensions: (3, 2|4)>
```

```python
>>> s.axes_manager[0].name = 'x'
>>> s.axes_manager[1].name = 'y'
>>> s.axes_manager[2].name = 't'
```

```python
>>> s.inav[0,0].data
array([0, 1, 2, 3])
```

```python
>>> s.inav[0,0].data
array([0, 1, 2, 3])
```

```python
>>> s.inav[0,0].axes_manager
Axes manager, axes: (4)
```

```
1.6. The Signal class
```
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.isig[0]
<BaseSignal, title: , dimensions: (2, 3)>
>>> s.isig[0].axes_manager
<Axes manager, axes: (3, 2)>
 Name | size | index | offset | scale | units
---------------- | ------ | ------ | ------- | ------- | -----
     x |        3 |        0 | 0 | 1 | <undefined>
     y |        2 |        0 | 0 | 1 | <undefined>
>>> s.isig[0].data
array([[ 0,  4,  8],
        [12, 16, 20]])
```

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

```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(), **, <<, >>, &, ^, |), 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:
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)>
```

### 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
>>> 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)):
```

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
>>> 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.

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:

```python
>>> 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.
HyperSpy Documentation, Release 1.6.0

```python
>>> 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.

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

```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]

>>> 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)
```

The execution can be sped up by passing `parallel` keyword to the `map()` method:

```python
>>> import time

>>> def slow_func(data):
...   time.sleep(1.)

(continues on next page)```
...    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]

Note: HyperSpy implements thread-based parallelism for the map() method. You can control the number of threads that are created by passing an integer value to the max_workers keyword argument. By default, it will use \text{min}(32, \text{os.cpu_count})

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 \text{Indexing}. In addition it can be performed using the following method or GUIs if cropping signal1D or signal2D. There is also a general \text{crop}() method that operates in place.

Rebinning

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

The \text{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 \text{lazy-evaluation} and is usually faster. Otherwise, the operation requires linear interpolation.

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

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

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

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

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

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

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()
```

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

```python
>>> image = image.split()[0].split()[0]
>>> image.plot()
```
Fig. 9: Stacking example.
Fig. 10: Splitting example.
Fast Fourier Transform (FFT)

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

```python
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft_shifted = im.fft(shift=True)
>>> fft_shifted.plot()
```

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

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

```python
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft = im.fft(True)
>>> fft.plot(True, vmin="30th")
```

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

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

```python
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft = im.fft(True)
>>> fft.plot(True, vmin="30th")
```

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

```python
>>> im = hs.datasets.example_signals.reference_hologram()
>>> fft = im.fft(shift=True)
>>> fft_apodized = im.fft(shift=True, apodization=True)
>>> fft_apodized.plot(True, vmin="30th")
```

**Inverse Fast Fourier Transform (iFFT)**

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

```python
>>> im_ifft = im.fft(shift=True).ifft(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) 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
>>> 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")
```
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 function `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, 3)) # 3D signal
>>> s4 = hs.signals.BaseSignal(np.random.rand(4, 4, 4, 4)) # 4D signal
>>> hs.plot.plot_images(hs.transpose(s2, s3, s4, signal_axes=2))
```

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

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

```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'`).
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)
```

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

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

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

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

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
General
title = Signal
    binned = False
    signal_type =

>>> s.estimate_poissonian_noise_variance()
>>> s.metadata
General
title = Signal
```
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 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:
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 2\(\pi\) 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].

**Calculate and display Argand diagram**

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

An example of calculation of Aragand diagram is shown for electron holography data.

**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 Interactive Operations and Region of Interest (ROI)

#### 1.7.1 Interactive operations

The function *interactive()* simplifies the definition of operations that are automatically updated when an event is triggered. By default the operation is recomputed when the data or the axes of the original signal is changed.
Interactive operations can be performed in a chain.

```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.],
       [ 48.,  51.,  54.,  57.],
       [ 66.,  69.,  72.,  75.],
       [ 84.,  87.,  90.,  93.]]
>>> 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]])
```

1.7.2 Region Of Interest (ROI)

ROIs can be defined to select part of any compatible signal and may be applied either to the navigation or to the signal axes. A number of different ROIs are available:

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

Once created, an ROI can be applied to the signal:

```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 example shows how to interactively apply ROIs to an image. Note that it is necessary to plot the signal onto which the widgets will be added before calling `interactive()`.

```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)
```

(continues on next page)
>>> point_roi = hs.roi.Point2DROI(256, 256)
>>> im.plot()
>>> roi2D = rectangular_roi.interactive(im, color="blue")
>>> roi1D = line_roi.interactive(im, color="yellow")
>>> roi0D = point_roi.interactive(im, color="red")

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

```python
>>> import scipy.misc
>>> 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.
1.7. Interactive Operations and Region of Interest (ROI)
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 \texttt{signal\_range} argument.

All ROIs have a \texttt{gui()} method that displays an user interface if a hyperspy GUI is installed (currently only works with the \texttt{hyperspy\_gui\_ipywidgets} GUI), enabling precise control of the ROI parameters:

\begin{verbatim}
>>> # continuing from above:
... >>> roi.gui()
\end{verbatim}

New in version 1.4: \texttt{angle()} can be used to calculate an angle between ROI line and one of the axes providing its name through optional argument \texttt{axis}:

\begin{verbatim}
>>> import scipy
... >>> holo = hs.datasets.example_signals.object_hologram()
... >>> roi = hs.roi.Line2DROI(x1=465.577, y1=445.15, x2=169.4, y2=387.731, linewidth=0)
... >>> holo.plot()
... >>> ss = roi.interactive(holo)
... >>> roi.angle(axis='y')
-100.97166759025453
\end{verbatim}

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

\begin{verbatim}
>>> roi.angle(axis='vertical', units='radians')
-1.7622880506791903
\end{verbatim}

Conveniently, \texttt{angle()} can be used to rotate an image to align selected features with respect to vertical or horizontal axis:
Hologram of Fe needle Signal
>>> holo.map(scipy.ndimage.rotate, angle=roi.angle(axis='horizontal'), inplace=False).
˓→plot()

1.7.3 Slicing using ROIs

ROIs can be used in place of slices when indexing. For example:

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> roi = hs.roi.SpanROI(left=5, right=15)
>>> sc = s.isig[roi]
>>> 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.

New in version 1.6: New __getitem__() method for all ROIs.

In addition the following all ROIs have a py:meth:`__getitem__` method that enables using them in place of tuples. For example, the method `align2D()` takes a roi argument with the left, right, top, bottom coordinates of the ROI. Handily, we can pass a `RectangularROI` ROI instead.
>>> import hyperspy.api as hs
>>> import numpy as np
>>> im = hs.signals.Signal2D(np.random.random((10,30,30))
>>> roi = hs.roi.RectangularROI(left=2, right=10, top=0, bottom=5))
>>> tuple(roi)
(2.0, 10.0, 0.0, 5.0)
>>> im.align2D(roi=roi)

1.8 Signal1D Tools

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

1.8.1 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:

```python
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:

```python
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:

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

1.8.2 Background removal

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

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

Example of usage:

```python
s = hs.datasets.artificial_data.get_core_loss_eels_signal(add_powerlaw=True)
s.remove_background(zero_fill=False)
```
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.
1.8.3 Calibration

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

1.8.4 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()`

1.8.5 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.

1.8.6 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()`

1.8.7 Spike removal

`spikes_removal_tool()` provides an user interface to remove spikes from spectra. The derivative histogram allows to identify the appropriate threshold.

1.8.8 Peak finding

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

1.8.9 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()`
1.9 Signal2D Tools

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

### 1.9.1 Signal registration and alignment

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

```python
# Estimate shifts, then align the images
>>> shifts = s.estimate_shift2D()
>>> s.align2D(shifts=shifts)

# Estimate and align in a single step
>>> s.align2D()
```

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

Sub-pixel accuracy can be achieved in two ways:

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

```python
>>> shifts = s.estimate_shift2D(sub_pixel_factor=20)
```

# stat method

```python
>>> shifts = s.estimate_shift2D(reference="stat")
```

# combined upsampling and statistical method

```python
>>> shifts = s.estimate_shift2D(reference="stat", sub_pixel_factor=20)
```

If you have a large stack of images, you can perform the image alignment step in parallel by passing `parallel=True`. You can control the number of threads used with the `max_workers` argument. See the `map documentation` for more information.

```python
# Estimate shifts
>>> shifts = s.estimate_shift2D()

# Align images in parallel using 4 threads
>>> s.align2D(shifts=shifts, parallel=True, max_workers=4)
```

## 1.9.2 Cropping an image

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

```python
>>> 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:

```python
>>> 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:

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

## 1.9.3 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.
Fig. 16: Interactive image cropping using a ROI.

1.9. Signal2D Tools

---

Hologram of Fe needle signal

- x axis (nm)
- y axis (nm)
- Scale: 100 nm
- Colorbar: 6000 to -1000
1.9.4 Peak finding

New in version 1.6.

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

**Maximum based peak finder**

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

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

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

**Zaefferer peak finder**

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

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

**Ball statistical peak finder**

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

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

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

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

Template matching

```python
>>> x, y = np.meshgrid(np.arange(-2, 2.5, 0.5), np.arange(-2, 2.5, 0.5))
>>> template = hs.model.components2D.Gaussian2D().function(x, y)
>>> s.find_peaks(method='template_matching', template=template)
```

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

1.9.5 Interactive parametrization

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

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

Several widgets are available:

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

**Note:** Some methods take significantly longer than others, particularly where there are a large number of peaks to be found. The plotting window may be inactive during this time.
1.10 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.

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

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

1.10.1 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.

Fig. 18: Visualisation of a 1D spectrum image.
New in version 1.4: Customizable keyboard shortcuts to navigate multi-dimensional datasets.

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

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.  

![Fig. 19: Visualisation of a 2D spectrum image using two pointers.](image)

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

<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:

```
>>> 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')`.

### 1.10.2 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.](image)

New in version 1.4: `l` keyboard shortcut

The following keyboard shortcuts are available when the 2D signal figure is in focus:
Fig. 21: Visualisation of a 2D image stack.

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>

1.10.3 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')
```

New in version 1.4: norm keyword argument

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

New in version 1.6: autoscale keyword argument

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

- 'x': to reset the horizontal axes
- 'y': to reset the vertical axes
Fig. 22: Custom colormap and switched off scalebar in an image.
• 'v': to reset the contrast of the image according to vmin and vmax

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

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

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

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

```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, cmap="RdBu", tight_layout=True)
```

**Fig. 23:** 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)
```
1.10.4 Customizing the “navigator”

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

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

```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))
```

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

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

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)
```

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')
```

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)
```

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

(2013) (see abstract).
Fig. 26: Visualisation of a 3D image with a slider.

Fig. 27: Visualisation of a 3D spectrum image with sliders.
Fig. 28: Visualisation of a 3D spectrum image. The navigator is an external signal.

Fig. 29: Visualisation of a stack of 2D spectral images.
1.10.5 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))
```

Note: See also the SEM EDS tutorials.

Note: The sample and the data used in this chapter are described in P. Burdet, et al., Ultramicroscopy, 148, p. 158-167 (2015).
Fig. 31: Visualisation of isosurfaces with mayavi.
1.10.6 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(range(10,70,10))
>>> image.map(scipy.ndimage.rotate, angle=angles.T, reshape=False)
>>> hs.plot.plot_images(image, tight_layout=True)
```

![Fig. 32: Figure generated with plot_images() using the default values.](image)

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)
```

(continues on next page)
>>> angles = hs.signals.BaseSignal(range(10,70,10))
>>> image.map(scipy.ndimage.rotate, angle=angles.T, reshape=False)
>>> hs.plot.plot_images(
...   image, suptitle='Turning Ascent', axes_decor='off',
...   label=[f'Rotation \{i\}°'.format(angles.data[i]) for i in range(angles.data.shape[0])], colorbar=None)

Fig. 33: Figure generated with \texttt{plot\_images()} with customised labels.

\texttt{plot\_images()} can also be used to easily plot a list of \texttt{Images}, comparing different \texttt{Signals}, including RGB images. This example also demonstrates how to wrap labels using \texttt{labelwrap} (for preventing overlap) and using a single \texttt{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])
>>> 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=True, 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
```

(continues on next page)
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)

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

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 urlopen, urlretrieve
from zipfile import ZipFile
files = urlretrieve('https://www.dropbox.com/s/ecd1gwxjg04m5mx/HyperSpy_demos_EDS_TEM_files.zip?raw=1', . . .)
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).

```python
>>> 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', vmin='1th', vmax='99th', scalebar='all',
...                     scalebar_color='black', suptitle_fontsize=16,
...                     padding={'top':0.8, 'bottom':0.10, 'left':0.05,
...                             'right':0.85, 'wspace':0.20, 'hspace':0.10})
```

![X-ray line intensity of Core shell:](image)

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

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

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

```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', vmin='1th', vmax='99th', scalebar=[0],
...                     scalebar_color='white', suptitle_fontsize=16)
```

The `cmap` argument can also be given as 'mpl_colors', and as a result, the images will be plotted with col-
Fig. 36: Using `plot_images()` to plot the output of `get_lines_intensity()` using a unique colormap for each image.

...ormaps 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(Tr...
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”.

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)
...

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

Fig. 38: Figure generated by `plot_spectra()` using the overlap style.

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

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

Fig. 40: Customising the line colors in `plot_spectra()`.
A simple extension of this functionality is to customize the colormap that is used to generate the list of colors. Using a list comprehension, one can generate a list of colors that follows a certain colormap:

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

```python
>>> 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.](image)

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')
```
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:
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.
Plotting profiles interactively

Spectra or line profile can be plotted interactively on the same figure using the `plot_spectra()` function. For example, profiles obtained from different Signal2D using the `Line2DROI` ROI can be plotted interactively:

```python
>>> im0 = hs.datasets.example_signals.reference_hologram()
>>> im1 = hs.datasets.example_signals.object_hologram()
>>> im0.plot()
>>> im1.plot()
>>> # Create the ROI
>>> line_profile = hs.roi.Line2DROI(400, 250, 220, 600)
>>> # Obtain the signals to plot by "slicing" the signals with the ROI
>>> line0 = line_profile.interactive(im0)
>>> line1 = line_profile.interactive(im1)
>>> # Plotting the profile on the same figure
>>> hs.plot.plot_spectra([line0, line1])
```

Fig. 48: Plotting profiles from different images interactively.
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.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> hs.plot.plot_signals([s1, s2])
```

![Plot of two spectra with synced navigation](image)

Fig. 49: The `plot_signals()` plots several signals with optional synchronized navigation.

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

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

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

```python
>>> import scipy.misc
>>> s1 = hs.signals.Signal1D(scipy.misc.face()).as_signal1D(0).inav[:, :3]
>>> s2 = s1.deepcopy()*-1
>>> s3 = hs.signals.Signal1D(np.linspace(0,9,9).reshape([3,3]))
>>> hs.plot.plot_signals([s1, s2], navigator_list=['slider', s3])
```
Fig. 50: Customising the navigator in `plot_signals()`.

Fig. 51: Customising the navigator in `plot_signals()` by providing a navigator list.
Several signals can also be plotted without syncing the navigation by using sync=False. The navigator_list can still be used to specify a navigator for each plot:

```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. 52: Disabling synchronised navigation in `plot_signals()`.

### 1.10.7 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:

```python
>>> from skimage.feature import peak_local_max
>>> im = 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)
```

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:

```python
```
Fig. 53: Rectangle static marker.

Fig. 54: Point markers in image.
>>> 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  #middle of signal plot
>>> m = hs.plot.markers.text(x=x, y=s.isig[x].data+2,
...         text=[i for i in 'abcdefghij'])
>>> s.add_marker(m)

Fig. 55: 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:

>>> 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)>
>>> s.plot()

Fig. 56: Plotting with permanent markers.

Markers can be removed by deleting them from the metadata

>>> s = hs.signals.Signal2D(np.arange(100).reshape(10, 10))
>>> marker = hs.markers.point(5, 9)
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)
>>> 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. 57: 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)))
>>> s.add_marker(marker, permanent=True)
```

Fig. 58: 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, 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
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> markers = (hs.markers.point(tuple(random()*10 for i in range(3)),
...                           tuple(random()*10 for i in range(3)),
...                           size=30, color=np.random.rand(3,1))
...                        for i in range(500))
>>> s.add_marker(markers, permanent=True)
```

Fig. 59: 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
>>> s = hs.signals.Signal2D(np.arange(300).reshape(3, 10, 10))
>>> markers = []
>>> 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)),  
...         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))))
>>> s.add_marker(markers, permanent=True)
```

Fig. 60: 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))
>>> marker = hs.markers.point(2, 1, color='red')
>>> s.add_marker(marker, plot_marker=False, permanent=True)
>>> s.metadata.Markers
    point = <marker.Point, point (x=2,y=1,color=red,size=20)>
>>> s.save("storing_marker.hdf5")
>>> s1 = hs.load("storing_marker.hdf5")
>>> s1.metadata.Markers
    point = <hyperspy.drawing._markers.point.Point object at 0x7efcfadb06d8>
```

## 1.11 Machine learning

### 1.11.1 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.

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

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

1.11. Machine learning
1.11.2 Decomposition

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


glob >>> import numpy as np
>>> from hyperspy.signals import Signal1D

```python
>>> s = Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition()
```

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

**Note:** The signal \( s \) must be multi-dimensional, \textit{i.e.} \( s\text{.axes_manager.navigation_size} > 1 \)

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

To reconstruct your denoised or reduced model, run the \texttt{get_decomposition_model()} method. For example:


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

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

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

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


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

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

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


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

Some of the algorithms described below require \texttt{output_dimension} to be provided.
Available algorithms

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SVD&quot; (default)</td>
<td><code>svd_pca()</code></td>
</tr>
<tr>
<td>&quot;MLPCA&quot;</td>
<td><code>mlpca()</code></td>
</tr>
<tr>
<td>&quot;sklearn_pca&quot;</td>
<td><code>sklearn.decomposition.PCA</code></td>
</tr>
<tr>
<td>&quot;NMF&quot;</td>
<td><code>sklearn.decomposition.NMF</code></td>
</tr>
<tr>
<td>&quot;sparse_pca&quot;</td>
<td><code>sklearn.decomposition.SparsePCA</code></td>
</tr>
<tr>
<td>&quot;mini_batch_sparse_pca&quot;</td>
<td><code>sklearn.decomposition.MinibatchSparsePCA</code></td>
</tr>
<tr>
<td>&quot;RPCA&quot;</td>
<td><code>rpca_godec()</code></td>
</tr>
<tr>
<td>&quot;ORPCA&quot;</td>
<td><code>ORPCA</code></td>
</tr>
<tr>
<td>&quot;ORNMF&quot;</td>
<td><code>ORNMF</code></td>
</tr>
<tr>
<td>custom object</td>
<td>An object implementing <code>fit()</code> and <code>transform()</code> methods</td>
</tr>
</tbody>
</table>

**Singular value decomposition (SVD)**

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

```python
>>> import numpy as np
>>> from hyperspy.signals import Signal1D
>>> s = Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition()
```

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

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

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

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

If you have scikit-learn installed:

```python
>>> s.decomposition(algorithm="sklearn_pca")
```

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

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

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

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

```python
>>> from sklearn.decomposition import PCA

>>> s.decomposition(algorithm=PCA())
```

Poissonian noise

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

To apply Poissonian noise normalization to your data:

```python
>>> s.decomposition(normalize_poissonian_noise=True)

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

**Warning**: Poisson noise normalization cannot be used in combination with data centering using the 'centre' argument. Attempting to do so will raise an error.

Maximum likelihood principal component analysis (MLPCA)

Instead of applying Poisson noise normalization to your data, you can instead use an approach known as Maximum Likelihood PCA (MLPCA), which provides a more robust statistical treatment of non-Gaussian “heteroskedastic noise”.

```python
>>> s.decomposition(algorithm="MLPCA")
```

For more information, please read the method documentation for `mlpca()`.

**Note**: You must set the `output_dimension` when using MLPCA.
Robust principal component analysis (RPCA)

PCA is known to be very sensitive to the presence of outliers in data. These outliers can be the result of missing or dead pixels, X-ray spikes, or very low count data. If one assumes a dataset, $X$, to consist of a low-rank component $L$ corrupted by a sparse error component $S$, such that $X = L + S$, then Robust PCA (RPCA) can be used to recover the low-rank component for subsequent processing [Candes2011].

![Schematic diagram of the robust PCA problem](image)

**Fig. 61:** Schematic diagram of the robust PCA problem, which combines a low-rank matrix with sparse errors. Robust PCA aims to decompose the matrix back into these two components.

**Note:** You must set the `output_dimension` when using Robust PCA.

The default RPCA algorithm is GoDec [Zhou2011]. In HyperSpy it returns the factors and loadings of $L$. RPCA solvers work by using regularization, in a similar manner to lasso or ridge regression, to enforce the low-rank constraint on the data. The low-rank regularization parameter, $\lambda_1$, defaults to $1/\sqrt{n\text{features}}$, but it is strongly recommended that you explore the behaviour of different values.

```python
>>> s.decomposition(algorithm="RPCA", output_dimension=3, lambda1=0.1)
```

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, including the regularization parameter highlighted above. Again, it is strongly recommended that you explore the behaviour of these parameters. To further improve the convergence, you can “train” the algorithm with the first few samples of your dataset. For example, the following code will train ORPCA using the first 32 samples of the data.

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

Finally, online RPCA includes two alternatives methods to the default block-coordinate descent solver, which can again improve both the convergence and speed of the algorithm. These are particularly useful for very large datasets.

The methods are based on stochastic gradient descent (SGD), and take an additional parameter to set the learning rate. The learning rate dictates the size of the steps taken by the gradient descent algorithm, and setting it too large can lead to oscillations that prevent the algorithm from finding the correct minima. Usually a value between 1 and 2 works well:

```python
>>> s.decomposition(algorithm="RPCA",
... output_dimension=3,
... method="SGD",
... subspace_learning_rate=1.1)
```

You can also use Momentum Stochastic Gradient Descent (MomentumSGD), which typically improves the convergence properties of stochastic gradient descent. This takes the further parameter “momentum”, which should be a
fraction between 0 and 1.

```python
>>> s.decomposition(algorithm="RPCA",
...     output_dimension=3,
...     method="MomentumSGD",
...     subspace_learning_rate=1.1,
...     subspace_momentum=0.5)
```

Using the “SGD” or “MomentumSGD” methods enables the subspace, i.e. the underlying low-rank component, to be tracked as it changes with each sample update. The default method instead assumes a fixed, static subspace.

**Non-negative matrix factorization (NMF)**

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 [Nicoletti2013].

NMF takes the optional argument `output_dimension`, which determines the number of components to keep. Setting this to a small number is recommended to keep the computation time small. Often it is useful to run a PCA decomposition first and use the scree plot to determine a suitable value for `output_dimension`.

**Robust non-negative matrix factorization (RNMF)**

In a similar manner to the online, robust methods that complement PCA above, HyperSpy includes an online robust NMF method. This is based on the OPGD (Online Proximal Gradient Descent) algorithm of [Zhao2016].

**Note:** You must set the `output_dimension` when using Robust NMF.

As before, you can control the regularization applied via the parameter “lambda1”:

```python
>>> s.decomposition(algorithm="ORNMF", output_dimension=3, lambda1=0.1)
```

The MomentumSGD method is useful for scenarios where the subspace, i.e. the underlying low-rank component, is changing over time.

```python
>>> s.decomposition(algorithm="ORNMF",
...     output_dimension=3,
...     method="MomentumSGD",
...     subspace_learning_rate=1.1,
...     subspace_momentum=0.5)
```

Both the default and MomentumSGD solvers assume an $l2$-norm minimization problem, which can still be sensitive to very heavily corrupted data. A more robust alternative is available, although it is typically much slower.

```python
>>> s.decomposition(algorithm="ORNMF", output_dimension=3, method="RobustPGD")
```
Custom decomposition algorithms

HyperSpy supports passing a custom decomposition algorithm, provided it follows the form of a scikit-learn estimator. Any object that implements fit() and transform() methods is acceptable, including sklearn.pipeline, Pipeline and sklearn.model_selection.GridSearchCV. You can access the fitted estimator by passing return_info=True.

```python
>>> # Passing a custom decomposition algorithm
>>> from sklearn.preprocessing import MinMaxScaler
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.decomposition import PCA

>>> pipe = Pipeline(["scaler", MinMaxScaler()], "PCA", PCA())

>>> out = s.decomposition(algorithm=pipe, return_info=True)
```

1.11.3 Blind Source Separation

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

**Warning:** The BSS algorithms operate on the result of a previous decomposition analysis. It is therefore necessary to perform a decomposition first before calling blind_source_separation(). otherwise it will raise an error.

You must provide an integer number_of_components argument, or a list of components as the comp_list argument. This performs BSS on the chosen number/list of components from the previous decomposition.

To perform blind source separation on the result of a previous decomposition, run the blind_source_separation() method, for example:

```python
>>> import numpy as np
>>> from hyperspy.signals import Signal1D

>>> s = Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition(output_dimension=3)

>>> s.blind_source_separation(number_of_components=3)

# Perform only on the first and third components
>>> s.blind_source_separation(comp_list=[0, 2])
```
Available algorithms

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

Table 6: Available blind source separation algorithms in HyperSpy

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;sklearn_fastica&quot; (default)</td>
<td>sklearn.decomposition.FastICA</td>
</tr>
<tr>
<td>&quot;orthomax&quot;</td>
<td>orthomax()</td>
</tr>
<tr>
<td>&quot;FastICA&quot;</td>
<td>mdp.nodes.FastICANode</td>
</tr>
<tr>
<td>&quot;JADE&quot;</td>
<td>mdp.nodes.JADENode</td>
</tr>
<tr>
<td>&quot;CuBICA&quot;</td>
<td>mdp.nodes.CuBICANode</td>
</tr>
<tr>
<td>&quot;TDSEP&quot;</td>
<td>mdp.nodes.TDSEPNode</td>
</tr>
<tr>
<td>custom object</td>
<td>An object implementing fit() and transform() methods</td>
</tr>
</tbody>
</table>

Note: Except orthomax(), all of the implemented BSS algorithms listed above rely on external packages being available on your system. sklearn_fastica requires scikit-learn while FastICA, JADE, CuBICA, TDSEP require the Modular toolkit for Data Processing (MDP).

Orthomax

Orthomax rotations are a statistical technique used to clarify and highlight the relationship among factors, by adjusting the coordinates of PCA results. The most common approach is known as “varimax”, which intended to maximize the variance shared among the components while preserving orthogonality. The results of an orthomax rotation following PCA are often “simpler” to interpret than just PCA, since each component has a more discrete contribution to the data.

```python
>>> import numpy as np
>>> from hyperspy.signals import Signal1D

>>> s = Signal1D(np.random.randn(10, 10, 200))
>>> s.decomposition(output_dimension=3)

>>> s.blind_source_separation(number_of_components=3, algorithm="orthomax")
```
Independent component analysis (ICA)

One of the most common approaches for blind source separation is Independent Component Analysis (ICA). This separates a signal into subcomponents by assuming that the subcomponents are (a) non-Gaussian, and (b) that they are statistically independent from each other.

Custom BSS algorithms

As with decomposition, HyperSpy supports passing a custom BSS algorithm, provided it follows the form of a scikit-learn estimator. Any object that implements fit() and transform() methods is acceptable, including sklearn.pipeline.Pipeline and sklearn.model_selection.GridSearchCV. You can access the fitted estimator by passing return_info=True.

```python
>>> # Passing a custom BSS algorithm
>>> from sklearn.preprocessing import MinMaxScaler
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.decomposition import FastICA

>>> pipe = Pipeline(["scaler", MinMaxScaler()], ["ica", FastICA()])
>>> out = s.blind_source_separation(number_of_components=3, algorithm=pipe, return_info=True)

Pipeline(memory=None,
    steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
           ('ica', FastICA(algorithm='parallel', fun='logcosh', fun_args=None,
                           max_iter=200, n_components=3, random_state=None,
                           tol=0.0001, w_init=None, whiten=True))],
    verbose=False)
```

1.11.4 Cluster analysis

New in version 1.6.

Introduction

Cluster analysis or clustering is the task of grouping a set of measurements such that measurements in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). A HyperSpy signal can represent a number of large arrays of different measurements which can represent spectra, images or sets of parameters. Identifying and extracting trends from large datasets is often difficult and decomposition methods, blind source separation and cluster analysis play an important role in this process.

Cluster analysis, in essence, compares the “distances” (or similar metric) between different sets of measurements and groups those that are closest together. The features it groups can be raw data points, for example, comparing for every navigation dimension all points of a spectrum. However, if the dataset is large, the process of clustering can be computationally intensive so clustering is more commonly used on an extracted set of features or parameters. For example, extraction of two peak positions of interest via a fitting process rather than clustering all spectra points.

In favourable cases, matrix decomposition and related methods can decompose the data into a (ideally small) set of significant loadings and factors. The factors capture a core representation of the features in the data and the loadings provide the mixing ratios of these factors that best describe the original data. Overall, this usually represents a much smaller data volume compared to the original data and can helps to identify correlations.
A detailed description of the application of cluster analysis in x-ray spectro-microscopy and further details on the theory and implementation can be found in [Lerotic2004].

Nomenclature

Taking the example of a 1D Signal of dimensions $(20, 10|4)$ containing the dataset, we say there are 200 samples. The four measured parameters are the features. If we choose to search for 3 clusters within this dataset, we derive three main values:

1. The labels, of dimensions $(3| 20, 10)$. Each navigation position is assigned to a cluster. The labels of each cluster are boolean arrays that mark the data that has been assigned to the cluster with True.

2. The cluster_distances, of dimensions $(3| 20, 10)$, which are the distances of all the data points to the centroid of each cluster.

3. The “cluster signals”, which are signals that are representative of their clusters. In HyperSpy two are computer: cluster_sum_signals and cluster_centroid_signals, of dimensions $(3| 4)$, which are the sum of all the cluster signals that belong to each cluster or the signal closest to each cluster centroid respectively.

Clustering functions HyperSpy

All HyperSpy signals have the following methods for clustering analysis:

- `cluster_analysis()`  
- `plot_cluster_results()`  
- `plot_cluster_labels()`  
- `plot_cluster_signals()`  
- `plot_cluster_distances()`  
- `get_cluster_signals()`  
- `get_cluster_labels()`  
- `get_cluster_distances()`  
- `estimate_number_of_clusters()`  
- `plot_cluster_metric()`

The `cluster_analysis()` method can perform cluster analysis using any sklearn.clustering clustering algorithms or any other object with a compatible API. This involves importing the relevant algorithm class from scikit-learn.

```python
>>> from sklearn.cluster import KMeans
>>> s.cluster_analysis(cluster_source="signal", algorithm=KMeans(n_clusters=3, n_init=8))
```

For convenience, the default algorithm is `kmeans` algorithm and is imported internally. All extra keyword arguments are passed to the algorithm when present. Therefore the following code is equivalent to the previous one:

For example:

```python
>>> s.cluster_analysis(cluster_source="signal", n_clusters=3, preprocessing="norm", algorithm="kmeans", n_init=8)
```

is equivalent to:
cluster_analysis() computes the cluster labels. The clusters areas with identical label are averaged to create a set of cluster centres. This averaging can be performed on the signal itself, the bss or decomposition results or a user supplied signal.

Pre-processing

Cluster analysis measures the distances between features and groups them. It is often necessary to pre-process the features in order to obtain meaningful results.

For example, pre-processing can be useful to reveal clusters when performing cluster analysis of decomposition results. Decomposition methods decompose data into a set of factors and a set of loadings defining the mixing needed to represent the data. If signal 1 is reduced to three components with mixing 0.1 0.5 2.0, and signal 2 is reduced to a mixing of 0.2 1.0 4.0, it should be clear that these represent the same signal but with a scaling difference. Normalization of the data can again be used to remove scaling effects.

Therefore, the pre-processing step will highly influence the results and should be evaluated for the problem under investigation.

All pre-processing methods from (or compatible with) sklearn.preprocessing can be passed to the scaling keyword of the cluster_analysis() method. For convenience, the following methods from scikit-learn are available as standard: standard, minmax and norm as standard. Briefly, norm treats the features as a vector and normalizes the vector length. standard re-scales each feature by removing the mean and scaling to unit variance. minmax normalizes each feature between the minimum and maximum range of that feature.

Cluster signals

In HyperSpy cluster signals are signals that somehow represent their clusters. The concept is ill-defined, since cluster algorithms only assign data points to clusters. HyperSpy computes 2 cluster signals,

1. cluster_sum_signals, which are the sum of all the cluster signals that belong to each cluster.
2. cluster_centroid_signals, which is the signal closest to each cluster centroid.

When plotting the “cluster signals” we can select any of those above using the signal keyword argument:

```python
>>> s.plot_cluster_labels(signal="centroid")
```

In addition, it is possible to plot the mean signal over the different clusters:

```python
>>> s.plot_cluster_labels(signal="mean")
```

Clustering with user defined algorithms

User developed preprocessing or cluster algorithms can be used in place of the sklearn methods. A preprocessing object needs a fit_transform which appropriately scales the data. The example below defines a preprocessing class which normalizes the data then applies a square root to enhances weaker features.

```python
>>> class PowerScaling(object):
>>>     def __init__(self, power=0.5):
>>>         self.power = power
>>>     def fit_transform(self, self, data):
>>>         norm = np.amax(data, axis=1)
```
```python
>>> scaled_data = data/norm[:, None]
>>> scaled_data = scaled_data - np.min(scaled_data)+1.0e-8
>>> scaled_data = scaled_data ** self.power
>>> return scaled_data

The PowerScaling class can then be passed to the cluster_analysis method for use.

```python
>>> ps = PowerScaling()
>>> s.cluster_analysis(cluster_source="decomposition", number_of_components=3,
                      preprocessing=ps)

For user defined clustering algorithms the class must implementation fit and have a label_ attribute that contains the clustering labels. An example template would be:

```python
>>> class MyClustering(object):
>>>     def __init__(self):
>>>         self.labels_ = None
>>>     def fit_(self, X):
>>>         self.labels_ = do_something(X)

Examples

Clustering using decomposition results

Let’s use the make_blobs function supplied by scikit-learn to make dummy data to see how clustering might work in practice.

```python
>>> import hyperspy.api as hs
>>> from sklearn.datasets import make_blobs
>>> data = make_blobs(
...     n_samples=1000,
...     n_features=100,
...     centers=3,
...     shuffle=False,
...     random_state=1)[0].reshape(50, 20, 100)
>>> s = hs.signals.Signal1D(data)

```python
>>> hs.plot.plot_images(data.T)
To see how cluster analysis works it’s best to first examine the signal. Moving around the image you should be able to see 3 distinct regions in which the 1D signal modulates slightly.

```python
>>> s.plot()
```

Let’s perform SVD to reduce the dimensionality of the dataset by exploiting redundancies:

```python
>>> s.decomposition()
>>> s.plot_explained_variance_ratio()
```
From the scree plot we deduce that, as expected, that the dataset can be reduce to 3 components. Let’s plot their loadings:

```python
>>> s.plot_decomposition_loadings(comp_ids=3, axes_decor="off")
```
In the SVD loading we can identify 3 regions, but they are mixed in the components. Let’s perform cluster analysis of decomposition results, to find similar regions and the representative features in those regions. Notice that this dataset does not require any pre-processing for cluster analysis.

```python
>>> s.cluster_analysis(cluster_source="decomposition", number_of_components=3,
                      preprocessing=None)
```  
```python
>>> s.plot_cluster_labels(axes_decor="off")
```

To see what the labels the cluster algorithm has assigned you can inspect the `cluster_labels`:

```python
>>> s.learning_results.cluster_labels[0]
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
```

(continues on next page)
In this case we know there are 3 cluster, but for real examples the number of clusters is not known *a priori*. A number of metrics, such as elbow, Silhouette and Gap can be used to estimate the optimal number of clusters. The elbow method measures the sum-of-squares of the distances within a cluster and, as for the PCA decomposition, an “elbow” or point where the gains diminish with increasing number of clusters indicates the ideal number of clusters. Silhouette analysis measures how well separated clusters are and can be used to determine the most likely number of clusters. As the scoring is a measure of separation of clusters a number of solutions may occur and maxima in the scores are used to indicate possible solutions. Gap analysis is similar but compares the “gap” between the clustered data results and those from a randomly data set of the same size. The largest gap indicates the best clustering. The metric results can be plotted to check how well-defined the clustering is.

```python
>>> s.estimate_number_of_clusters(cluster_source="decomposition", metric="gap")
>>> s.plot_cluster_metric()
```
The optimal number of clusters can be set or accessed from the learning results

```python
>>> s.learning_results.number_of_clusters
3
```

Clustering using another signal as source

In this example we will perform clustering analysis on the position of two peaks. The signals containing the position of the peaks can be computed for example using *curve fitting*. Given an existing fitted model, the parameters can be extracted as signals and stacked. Clustering can then be applied as described previously to identify trends in the fitted results.

Let’s start by creating a suitable synthetic dataset.

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

>>> s_dummy = hs.signals.Signal1D(np.zeros((64, 64, 1000)))

>>> s_dummy.axes_manager.signal_axes[0].scale = 2e-3
>>> s_dummy.axes_manager.signal_axes[0].units = "eV"
>>> s_dummy.axes_manager.signal_axes[0].name = "energy"

>>> m = s_dummy.create_model()

>>> m.append(hs.model.components1D.GaussianHF(fwhm=0.2))
>>> m.append(hs.model.components1D.GaussianHF(fwhm=0.3))

>>> m.components.GaussianHF.centre.map["values"][32:, :] = .3 + .1
>>> m.components.GaussianHF.centre.map["values"][32:, :] = .7 + .1

>>> m.components.GaussianHF_0.centre.map["values"][32:] = m.components.GaussianHF.
centre.map["values"][32] * 2

>>> m.components.GaussianHF_0.centre.map["values"][32] = m.components.GaussianHF.
centre.map["values"][32] * 0.5
```

(continues on next page)
>>> for component in m:
...     component.centre.map["is_set"][:] = True
...     component.centre.map["values"][:] += np.random.normal(size=(64, 64)) * 0.01

>>> s = m.as_signal()

>>> stack = hs.stack([m.components.GaussianHF.centre.as_signal(),
>>>                   m.components.GaussianHF_0.centre.as_signal()])

>>> s.estimate_number_of_clusters(cluster_source=stack.T, preprocessing="norm")
2

>>> s.cluster_analysis(cluster_source=stack.T, source_for_centers=s, n_clusters=2,
...                     preprocessing="norm")

>>> s.plot_cluster_labels()

Let’s now perform cluster analysis on the stack and calculate the centres using the spectrum image. Notice that we don’t need to fit the model to the data because this is a synthetic dataset. When analysing experimental data you will need to fit the model first. Also notice that here we need to pre-process the dataset by normalization in order to reveal the clusters due to the proportionality relationship between the position of the peaks.
HyperSpy Documentation, Release 1.6.0

Cluster labels of fitted model

```python
>>> s.plot_cluster_signals(signal="mean")
```
Notice that in this case averaging or summing the signals of each cluster is not appropriate, since the clustering criterium is the ratio between the peaks positions. A better alternative is to plot the signals closest to the centroids:

```python
>>> s.plot_cluster_signals(signal="centroid")
```
1.11.5 Visualizing results

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

**Scree plots**

**Note:** Scree plots are only available for the "SVD" and "PCA" algorithms.

PCA will sort the components in the dataset in order of decreasing variance. It is often useful to estimate the dimensionality of the data by plotting the explained variance against the component index. This plot is sometimes called a scree plot. For most datasets, the values in a scree plot will decay rapidly, eventually becoming a slowly descending line.

To obtain a scree plot for your dataset, run the `plot_explained_variance_ratio()` method:

```
>>> s.plot_explained_variance_ratio(n=20)
```

The point at which the scree plot becomes linear (often referred to as the “elbow”) is generally judged to be a good estimation of the dimensionality of the data (or equivalently, the number of components that should be retained - see below). Components to the left of the elbow are considered part of the “signal”, while components to the right are considered to be “noise”, and thus do not explain any significant features of the data.
By specifying a threshold value, a cutoff line will be drawn at the total variance specified, and the components above this value will be styled distinctly from the remaining components to show which are considered signal, as opposed to noise. Alternatively, by providing an integer value for threshold, the line will be drawn at the specified component (see below).

Note that in the above scree plot, the first component has index 0. This is because Python uses zero-based indexing. To switch to a “number-based” (rather than “index-based”) notation, specify the xaxis_type parameter:

```python
>>> s.plot_explained_variance_ratio(n=20, threshold=4, xaxis_type='number')
```

The number of significant components can be estimated and a vertical line drawn to represent this by specifying vline=True. In this case, the “elbow” is found in the variance plot by estimating the distance from each point in the variance plot to a line joining the first and last points of the plot, and then selecting the point where this distance is largest.

If multiple maxima are found, the index corresponding to the first occurrence is returned. As the index of the first component is zero, the number of significant PCA components is the elbow index position + 1. More details about the elbow-finding technique can be found in [Satopää2011], and in the documentation for estimate_elbow_position().

These options (together with many others), can be customized to develop a figure of your liking. See the documentation of plot_explained_variance_ratio() for more details.

Sometimes it can be useful to get the explained variance ratio as a spectrum. For example, to plot several scree plots obtained with different data pre-treatments in the same figure, you can combine plot_spectra() with get_explained_variance_ratio().
Fig. 63: PCA scree plot with number-based axis labeling and a threshold value specified.

The “elbow” is the point which is furthest from the line joining first and last points.

The no of significant PCA components is the elbow position + 1.
Fig. 64: PCA scree plot with number-based axis labeling and an estimate of the no of significant positions based on the “elbow” position

**Decomposition plots**

HyperSpy provides a number of methods for visualizing the factors and loadings found by a decomposition analysis. To plot everything in a compact form, use `plot_decomposition_results()`.

You can also plot the factors and loadings separately using the following methods. It is recommended that you provide the number of factors or loadings you wish to visualise, since the default is to plot all of them.

- `plot_decomposition_factors()`
- `plot_decomposition_loadings()`
Blind source separation plots

Visualizing blind source separation results is much the same as decomposition. You can use `plot_bss_results()` for a compact display, or instead:

- `plot_bss_factors()`
- `plot_bss_loadings()`

Clustering plots

Visualizing cluster results is much the same as decomposition. You can use `plot_bss_results()` for a compact display, or instead:

- `plot_cluster_results()`.
- `plot_cluster_signals()`.
- `plot_cluster_labels()`.
1.11.6 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()`

1.11.7 Saving and loading results

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.npz')

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

Exporting in different formats

You can also export the results of a machine learning analysis to any format supported by HyperSpy with the following methods:

- `export_decomposition_results()`
- `export_bss_results()`

These methods accept many arguments to customise the way in which the data is exported, so please consult the method documentation. The options include the choice of file format, the prefixes for loadings and factors, saving figures instead of data and more.

**Warning:** Data exported in this way cannot be easily loaded into HyperSpy’s machine learning structure.
1.12 Model fitting

HyperSpy can perform curve fitting of one-dimensional signals (spectra) and two-dimensional signals (images) in \( n \)-dimensional data sets. Models are defined by adding individual functions (components in HyperSpy’s terminology) to a `BaseModel` instance. Those individual components are then summed to create the final model function that can be fitted to the data, by adjusting the free parameters of the individual components.

Models can be created and fit to experimental data in both one and two dimensions i.e. spectra and images respectively. Most of the syntax is identical in either case. A one-dimensional model is created when a model is created for a `Signal1D` whereas a two-dimensional model is created for a `Signal2D`.

**Note:** Plotting and analytical gradient-based fitting methods are not yet implemented for the `Model2D` class.

1.12.1 Caveats

- Before creating a model verify that the **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.
- When importing data that has been binned using other software, in particular Gatan’s DM, the stored values may be the averages of the binned channels or pixels, instead of their sum, as would be required for proper statistical analysis. We therefore cannot guarantee that the statistics will be valid, and so strongly recommend that all pre-fitting binning is performed using Hyperspy.

1.12.2 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.

1.12.3 Model components

In HyperSpy a model consists of a sum of individual components. For convenience, HyperSpy provides a number of pre-defined model components as well as mechanisms to create your own components.
Pre-defined model components

Various components are available in one (components1d) and two-dimensions (components2d) to construct a model.

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

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

The following components developed with specific signal types in mind are currently available for one-dimensional models:

- EELSArctan
- DoublePowerLaw
- EELSCLEdge
- PESCoreLineShape
- PESVoigt
- SEE
- Vignetting

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

- Expression
- Gaussian2D

However, this doesn’t mean that you have to limit yourself to this meagre list of functions. As discussed below, it is very easy to turn a mathematical, fixed-pattern or Python function into a component.
Define components from a mathematical expression

The easiest way to turn a mathematical expression into a component is using the `Expression` component. For example, the following is all you need to create a `Gaussian` component with more sensible parameters for spectroscopy than the one that ships with HyperSpy:

```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",)
```

Define new components from a Python function

Of course `Expression` is only useful for analytical functions. You can define more general components modifying the following template to suit your needs:

```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'))

        # Optionally we can set the initial values
        self.parameter_1.value = parameter_1
        self.parameter_1.value = parameter_1

(continues on next page)```
Define components from a fixed-pattern

The `ScalableFixedPattern` component enables fitting a pattern (in the form of a `Signal1D` instance) to data by shifting (`shift`) and scaling it in the x and y directions using the `xscale` and `yscale` parameters respectively.

1.12.4 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>
```

```
Note: Sometimes components may be created automatically. For example, if the `Signal1D` is recognised as EELS data, a power-law background component may automatically be added to the model. Therefore, the table above may not all be empty on model creation.

To add a component to the model, first we have to create an instance of the component. Once the instance has been created we can add the component to the model using the `append()` and `extend()` methods for one or more components respectively.

As an example, let’s add several Gaussian components to the model:

```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></th>
<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>
<td>Gaussian</td>
</tr>
</tbody>
</table>
>>> gaussian2 = hs.model.components1D.Gaussian()  # Create another gaussian
>>> gaussian3 = hs.model.components1D.Gaussian()  # Create a third gaussian
We could use the `append()` method twice to add the two Gaussians, but when adding multiple components it is handier to use the `extend` method that enables adding a list of components at once.

```python
>>> m.extend((gaussian2, gaussian3))  # note the double parentheses!
>>> m.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>Gaussian</td>
<td>Gaussian</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Gaussian_0</td>
<td>Gaussian_0</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian_1</td>
<td>Gaussian_1</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'
>>> m.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>Gaussian</td>
<td>Carbon</td>
<td>Gaussian</td>
</tr>
<tr>
<td>1</td>
<td>Gaussian_0</td>
<td>Long Hydrogen name</td>
<td>Gaussian</td>
</tr>
<tr>
<td>2</td>
<td>Gaussian_1</td>
<td>Nitrogen</td>
<td>Gaussian</td>
</tr>
</tbody>
</table>
```

Notice that 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></th>
<th>Attribute Name</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
</table>
| (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
# | Attribute Name | Component Name | Component Type
---- | --------------------- | --------------------- | -------------------
0 | Carbon | Carbon | Gaussian
1 | Long_Hydrogen_name | Long Hydrogen name | Gaussian
2 | Nitrogen | Nitrogen | Gaussian
```

```python
>>> m[0]
<Carbon (Gaussian component)>
>>> m["Long Hydrogen name"]
<Long Hydrogen name (Gaussian component)>
```

It is possible to “switch off” a component by setting its `active` attribute to `False`. When a component is switched off, to all effects it is as if it was not part of the model. To switch it back on simply set the `active` attribute back to `True`.

In multi-dimensional signals it is possible to store the value of the `active` attribute at each navigation index. To enable this feature for a given component set the `active_is_multidimensional` attribute to `True`.

```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)
>>> g1.active_is_multidimensional = False
>>> g1._active_array is None
True
```
1.12.5 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)>
```

1.12.6 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 values of a particular component can be printed using the `print_current_values()` method.

```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
```

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()
```
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:
>>> 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
(<Parameter A of Carbon component>, <Parameter sigma of Carbon component>)
>>> m.print_current_values(only_free=True, fancy=False) # Print the values of all free parameters.

Model1D:
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
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

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

>>> gaussian3.A.value = 5 # Set the gaussian1 centre value to 5

Gaussian: Long Hydrogen name
Active:
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

Deprecated since version 1.2.0: Setting the twin_function and twin_inverse_function attributes. Set the twin_function_expr and twin_inverse_function_expr attributes instead.
New in version 1.2.0: `twin_function_expr` and `twin_inverse_function_expr`.

By default the coupling function is the identity function. However it is possible to set a different coupling function by setting the `twin_function_expr` and `twin_inverse_function_expr` attributes. For example:

```python
>>> gaussian2.A.twin_function_expr = "x**2"
>>> gaussian2.A.twin_inverse_function_expr = "sqrt(abs(x))"
>>> gaussian2.A.value = 4
>>> gaussian2.print_current_values(fancy=False)
Gaussian: Nitrogen
Active: True
Parameter Name | Free | Value | Std | Min | Max
============== | ===== | ========== | ========== | ========== | =========
A | True | 2.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None
centre | True | 0.0 | None | None | None
```

```python
>>> gaussian3.A.value = 4
>>> gaussian2.print_current_values(fancy=False)
Gaussian: Long Hydrogen name
Active: True
Parameter Name | Free | Value | Std | Min | Max
============== | ===== | ========== | ========== | ========== | =========
A | False | 16.0 | None | 0.0 | None
sigma | True | 1.0 | None | None | None
centre | True | 0.0 | None | None | None
```

### 1.12.7 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()`. HyperSpy implements a number of different optimization approaches, each of which can have particular benefits and/or drawbacks depending on your specific application. A good approach to choosing an optimization approach is to ask yourself the question “Do you want to...”:

- Apply bounds to your model parameter values?
- Use gradient-based fitting algorithms to accelerate your fit?
- Estimate the standard deviations of the parameter values found by the fit?
- Fit your data in the least-squares sense, or use another loss function?
- Find the global optima for your parameters, or is a local optima acceptable?

#### Optimization algorithms

The following table summarizes the features of some of the optimizers currently available in HyperSpy, including whether they support parameter bounds, gradients and parameter error estimation. The “Type” column indicates whether the optimizers find a local or global optima.
Table 7: Features of supported curve-fitting optimizers.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Bounds</th>
<th>Gradients</th>
<th>Errors</th>
<th>Loss function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lm&quot; (default)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Only &quot;ls&quot;</td>
<td>local</td>
</tr>
<tr>
<td>&quot;trf&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Only &quot;ls&quot;</td>
<td>local</td>
</tr>
<tr>
<td>&quot;dogbox&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Only &quot;ls&quot;</td>
<td>local</td>
</tr>
<tr>
<td>&quot;odr&quot;</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Only &quot;ls&quot;</td>
<td>local</td>
</tr>
<tr>
<td>scipy.optimize.minimize()</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>All</td>
<td>local</td>
</tr>
<tr>
<td>&quot;Differential Evolution&quot;</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>All</td>
<td>global</td>
</tr>
<tr>
<td>&quot;Dual Annealing&quot; **</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>All</td>
<td>global</td>
</tr>
<tr>
<td>&quot;SHGO&quot; **</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>All</td>
<td>global</td>
</tr>
</tbody>
</table>

Note: * All of the fitting algorithms available in scipy.optimize.minimize() are currently supported by HyperSpy; however, only some of them support bounds and/or gradients. For more information, please see the SciPy documentation.

** Requires scipy >= 1.2.0.

The default optimizer in HyperSpy is "lm", which stands for the Levenberg-Marquardt algorithm. In earlier versions of HyperSpy (< 1.6) this was known as "leastsq".

Loss functions

HyperSpy supports a number of loss functions. The default is "ls", i.e. the least-squares loss. For the vast majority of cases, this loss function is appropriate, and has the additional benefit of supporting parameter error estimation and goodness-of-fit testing. However, if your data contains very low counts per pixel, or is corrupted by outliers, the "ML-poisson" and "huber" loss functions may be worth investigating.

Least squares with error estimation

The following example shows how to perform least squares optimization with error estimation. First we create data consisting of a line \( y = a \times x + b \) with \( a = 1 \) and \( b = 100 \), and we then add Gaussian 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 **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()
```

Once the fit is complete, the optimized value of the parameters and their estimated standard deviation are stored in the following line attributes:

```latex
\begin{verbatim}
>>> line.a.value
0.9924615648843765
>>> line.b.value
103.67507406125888
>>> line.a.std
0.11771053738516088
\end{verbatim}
```

(continues on next page)
>>> line.b.std
13.541061301257537

**Warning:** When the noise is heteroscedastic, only if the `metadata.Signal.Noise_properties.variance` attribute of the `Signal1D` instance is defined can the parameter standard deviations be estimated accurately.

If the variance is not defined, the standard deviations are still computed, by setting variance equal to 1. However, this calculation will not be correct unless an accurate value of the variance is provided. See *Setting the noise properties* for more information.

## Weighted least squares with error estimation

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

```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()
```

```python
line.coefficients.value
(1.0052331707848698, -1.0723588390873573)
```

```python
line.coefficients.std
(0.0081710549764721901, 1.4117294994070277)
```

Because the noise is heteroscedastic, the least squares optimizer estimation is biased. A more accurate result can be obtained with weighted least squares, where the weights are proportional to the inverse of the noise variance. Although this is still biased for Poisson noise, it is a good approximation in most cases where there are a sufficient number of counts per pixel.

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

```python
line.coefficients.value
(1.0004224896604759, -0.46982916592391377)
```

```python
line.coefficients.std
(0.0055752036447948173, 0.46950832982673557)
```

**Warning:** When the attribute `metadata.Signal.Noise_properties.variance` is defined, the behaviour is to perform a weighted least-squares fit using the inverse of the noise variance as the weights. In this scenario, to then disable weighting, you will need to *unset* the attribute. You can achieve this with `set_noise_variance()`:

```python
>>> m.signal.set_noise_variance(None)
```

```python
>>> m.fit()  # This will now be an unweighted fit
```

```python
line.coefficients.value
(1.0052331707848698, -1.0723588390873573)
```
Poisson maximum likelihood estimation

To avoid biased estimation in the case of data corrupted by Poisson noise with very few counts, we can use Poisson maximum likelihood estimation (MLE) instead. This is an unbiased estimator for Poisson noise. To perform MLE, we must use a general, non-linear optimizer from the table above, such as Nelder-Mead or L-BFGS-B:

```python
>>> m.fit(optimizer="Nelder-Mead", loss_function="ML-poisson")
(1.0030718094185611, -0.63590210946134107)
```

Estimation of the parameter errors is not currently supported for Poisson maximum likelihood estimation.

Huber loss function

HyperSpy also implements the Huber loss function, which is typically less sensitive to outliers in the data compared to the least-squares loss. Again, we need to use one of the general non-linear optimization algorithms:

```python
>>> m.fit(optimizer="Nelder-Mead", loss_function="huber")
```

Estimation of the parameter errors is not currently supported for the Huber loss function.

Custom loss functions

As well as the built-in loss functions described above, a custom loss function can be passed to the model:

```python
>>> def my_custom_function(model, values, data, weights=None):
...     
...     # 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:
```

(continues on next page)
... weights = 1
... difference = (data - cur_value) * weights
... # Return squared and summed weighted difference
... return (difference**2).sum()

>>> # We must use a general non-linear optimizer
>>> m.fit(optimizer='Nelder-Mead', loss_function=my_custom_function)

If the optimizer requires an analytical gradient function, it can be similarly passed, using the following signature:

```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
... optimizer (that are not yet stored in the model).
... data : np.ndarray
... A one-dimensional array with current data that is being fitted.
... weights : {np.ndarray, None}
... An optional one-dimensional array with parameter weights.

... Returns
... -------
... gradients : np.ndarray
... a one-dimensional array of gradients, the size of `values`,
... containing each parameter gradient with the given values

... 
... # As an example, estimate maximum likelihood gradient:
... model.fetch_values_from_array(values)
... cur_value = model(onlyactive=True)

... # We use in-built jacobian estimation
... jac = model._jacobian(values, data)

... return -(jac * (data / cur_value - 1)).sum(1)
```

```python
>>> # We must use a general non-linear optimizer again
>>> m.fit(optimizer='L-BFGS-B',
... loss_function=my_custom_function,
... grad=my_custom_gradient_function)
```

Using gradient information

New in version 1.6: grad="analytical" and grad="fd" keyword arguments

Optimization algorithms that take into account the gradient of the loss function will often out-perform so-called “derivative-free” optimization algorithms in terms of how rapidly they converge to a solution. HyperSpy can use analytical gradients for model-fitting, as well as numerical estimates of the gradient based on finite differences.

If all the components in the model support analytical gradients, you can pass grad="analytical" in order to use this information when fitting. The results are typically more accurate than an estimated gradient, and the optimization often runs faster since fewer function evaluations are required to calculate the gradient.
Following the above examples:

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

>>> # Use a 2-point finite-difference scheme to estimate the gradient
>>> m.fit(grad="fd", fd_scheme="2-point")

>>> # Use the analytical gradient
>>> m.fit(grad="analytical")

>>> # Huber loss and Poisson MLE functions
>>> # also support analytical gradients
>>> m.fit(gradient="analytical", loss_function="huber")
>>> m.fit(gradient="analytical", loss_function="ML-poisson")
```

**Note:** Analytical gradients are not yet implemented for the `Model2D` class.

### Bounded optimization

Non-linear optimization can sometimes fail to converge to a good optimum, especially if poor starting values are provided. Problems of ill-conditioning and non-convergence can be improved by using bounded optimization.

All components’ parameters have the attributes `parameter.bmin` and `parameter.bmax` (“bounded min” and “bounded max”). When fitting using the `bounded=True` argument by `m.fit(bounded=True)` or `m.multifit(bounded=True)`, these attributes set the minimum and maximum values allowed for `parameter.value`.

Currently, not all optimizers support bounds - see the table above. In the following example, a Gaussian histogram is fitted using a `Gaussian` component using the Levenberg-Marquardt (“lm”) optimizer and bounds on the `centre` parameter.

```python
>>> s = hs.signals.BaseSignal(np.random.normal(loc=10, scale=0.01,
... size=100000)).get_histogram()
>>> s.metadata.Signal.binned = True
>>> m = s.create_model()
>>> g1 = hs.model.components1D.Gaussian()
>>> m.append(g1)
>>> g1.centre.value = 7
>>> g1.centre.bmin = 7
>>> g1.centre.bmax = 14
>>> m.fit(optimizer="lm", bounded=True)
>>> m.print_current_values(fancy=False)
```

```
Model1D: histogram
Gaussian: Gaussian
Active: True
Parameter Name | Free | Value | Std | Min | Max
--------------- | ----- | --------- | ---- | ---- | ----
A | True | 99997.3481 | 232.333693 | 0.0 | None
sigma | True | 0.00999184 | 2.68064163 | None | None
centre | True | 9.99980788 | 2.68064070 | 7.0 | 14.0
```

1.12. Model fitting 149
Optimization results

After fitting the model, details about the optimization procedure, including whether it finished successfully, are returned as `scipy.optimize.OptimizeResult` object, according to the keyword argument `return_info=True`. These details are often useful for diagnosing problems such as a poorly-fitted model or a convergence failure. You can also access the object as the `fit_output` attribute:

```python
>>> m.fit()
<scipy.optimize.OptimizeResult object>
```

```python
>>> type(m.fit_output)
<scipy.optimize.OptimizeResult object>
```

You can also print this information using the `print_info` keyword argument:

```python
# Print the info to stdout
>>> m.fit(optimizer="L-BFGS-B", print_info=True)
Fit info:
  optimizer=L-BFGS-B
  loss_function=ls
  bounded=False
  grad="fd"
Fit result:
  hess_inv: <3x3 LbfgsInvHessProduct with dtype=float64>
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
  nfev: 168
  nit: 32
  njev: 42
  status: 0
  success: True
  x: array([ 9.97614503e+03, -1.10610734e-01, 1.98380701e+00])
```

Goodness of fit

The chi-squared, reduced chi-squared and the degrees of freedom are computed automatically when fitting a (weighted) least-squares model (i.e. only when `loss_function="ls"`). They are stored as signals, in the `chisq`, `red_chisq` and `dof` attributes of the model respectively.

**Warning:** Unless `metadata.Signal.Noise_properties.variance` contains an accurate estimation of the variance of the data, the chi-squared and reduced chi-squared will not be computed correctly. This is true for both homoscedastic and heteroscedastic 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 passed to the `plot()` method of the corresponding signal object. For example, the following plots the model signal figure but not its navigator:

```python
g>>> 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 optimization often requires setting sensible starting parameters. This can be done by plotting the model and adjusting the parameters by hand.

Changed in version 1.3: All `notebook_interaction()` methods renamed to `gui()`. The `notebook_interaction()` methods 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`.

![Interactive widgets](image)

```
>>> n = s.create_model()
n.append(s.model.components.Gaussian())
n.append(s.model.components.Polynomial(5))
n.append(s.model.components.FIELDSWide('C_K'))
n.append(s.model.components.ScableFixedPattern(s.deepcopy()))
n.notebook_interaction()
```

Fig. 65: Interactive widgets for the full model in a Jupyter notebook. Drag the sliders to adjust current parameter values. Typing different minimum and maximum values changes the boundaries of the slider.

Also, `enable_adjust_position()` provides an interactive way of setting the position of the components with a well-defined position. `disable_adjust_position()` disables the tool.
Fig. 66: Interactive component position adjustment tool. Drag the vertical lines to set the initial value of the position parameter.
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()`:

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

`multifit()` fits the model at the first position, stores the result of the fit internally and move to the next position until reaching the end of the dataset.

**Note:** Sometimes this method can fail, especially in the case of a TEM spectrum image of a particle surrounded by vacuum (since in that case the top-left pixel will typically be an empty signal).

To get sensible starting parameters, you can do a single `fit()` after changing the active position within the spectrum image (either using the plotting GUI or by directly modifying `s.axes_manager.indices` as in Setting axis properties).

After doing this, you can initialize the model at every pixel to the values from the single pixel fit using `m.assign_current_values_to_all()`, and then use `multifit()` to perform the fit over the entire spectrum image.

New in version 1.6: New optional fitting iteration path “serpentine”

Typically, curve fitting on a multidimensional dataset happens in the following manner: Pixels are fit along the row from the first index in the first row, and once the final pixel is reached, one proceeds from the first index in the second row. Since the fitting procedure typically uses the fit of the previous pixel as the starting point for the next, a common problem with this fitting iteration path is that the fitting fails going from the end of one row to the beginning of the next, as the spectrum can change abruptly. This kind of iteration path is the default in HyperSpy (but will change to 'serpentine' in HyperSpy version 2.0). It can be explicitly set using the `multifit()` `iterpath='flyback'` argument.

A simple solution to the flyback fitting problem is to iterate through the signal indices in a horizontal serpentine pattern, as seen on the image below. This alternate iteration method can be enabled by the `multifit()` `iterpath='serpentine'` argument. The serpentine pattern supports n-dimensional navigation space, so the first index in the second frame of a three-dimensional navigation space will be at the last position of the previous frame.

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.

1.12.8 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 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
         dimensions = (|100)

>>> m1 = s.models.restore('myname')
>>> m1.components
   # | Attribute Name | Component Name | Component Type
     | ------------------- | -------------------- | --------------------
   0 | Lorentzian | Lorentzian | Lorentzian
```

### 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.

1.12.9 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()`

1.12.10 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 SAMFfire 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 from global statistical measures. These values are all attempted in order until a satisfactory result is found (not necessarily testing all available starting guesses). As a result, on average each pixel requires significantly more computations when compared to a local strategy.

More information about the exact procedure will be available once the accompanying paper is published.

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 to 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:

```plaintext
A | # | Strategy
-- | ---- | -------------------------
| 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(optimizer='lm', bounded=True)
```

### 1.13 Electron Energy Loss Spectroscopy

#### 1.13.1 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).

An utility function `get_edges_near_energy()` can be helpful to identify possible elements in the sample. `get_edges_near_energy()` returns a list of edges arranged in the order closest to the specified energy within a window, both measured in eV. The size of the window can be controlled by the argument `width` (default as 10)—If the specified energy is 849 eV and the width is 6 eV, it returns a list of edges with onset energy between 846 eV to 852 eV and they are arranged in the order closest to 849 eV.

```python
>>> from hyperspy.misc.eels.tools import get_edges_near_energy

>>> get_edges_near_energy(532)
['O_K', 'Pd_M3', 'Sb_M5', 'Sb_M4']

>>> get_edges_near_energy(849, width=6)
['La_M4', 'Fe_L1']
```

The static method `print_edges_near_energy()` in `EELSSpectrum` will print out a table containing more information about the edges.

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

>>> s.print_edges_near_energy(401, width=20)
+-------+-------------------+-----------+-----------------------------+
<table>
<thead>
<tr>
<th>edge</th>
<th>onset energy (eV)</th>
<th>relevance</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_K</td>
<td>401.0</td>
<td>Major</td>
<td>Abrupt onset</td>
</tr>
</tbody>
</table>
```

(continues on next page)
The method `edges_at_energy()` allows inspecting different sections of the signal for interactive edge identification (the default). A region can be selected by dragging the mouse across the signal and after clicking the Update button, edges with onset energies within the selected energy range will be displayed. By toggling the edge buttons, it will put or remove the corresponding edges on the signal. When the Complementary edge box is ticked, edges outside the selected range with the same element of edges within the selected energy range will be shown as well to aid identification of edges.

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

![Artificial core loss EEL spectrum Signal](image)

**Fig. 68:** Labels of edges can be put or remove by toggling the edge buttons.
Thickness estimation

New in version 1.6: Option to compute the absolute thickness, including the angular corrections and mean free path estimation.

The `estimate_thickness()` method can estimate the thickness from a low-loss EELS spectrum using the log-ratio method. If the beam energy, collection angle, convergence angle and sample density are known, the absolute thickness is computed using the method in [Iakoubovskii2008]. This includes the estimation of the inelastic mean free path (iMFP). For more accurate results, it is possible to input the iMFP of the material if known. If the density and/or the iMFP are not known, the output is the thickness relative to the (unknown) iMFP without any angular corrections.

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_oahver()`.

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.

1.13.2 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 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:

```python
>>> m.components
```

Conveniently, all the EELS core-loss components of the added elements are added automatically, names after its element symbol.

```python
>>> m.components.N_K
<N_K (EELSCLEdge component)>
```

(continues on next page)
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.

```
>>> m.enable_fine_structure()
```

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

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

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

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

**Note:** `m.smart_fit()` and `m.multifit(kind=’smart’)` are methods specific to the EELS model. The fitting procedure acts in iterative manner along the energy-loss-axis. First it fits only the background up to the first edge. It continues by deactivating all edges except the first one, then performs the fit. Then it only activates the the first two, fits, and repeats this until all edges are fitted simultaneously.

Other, non-EELSCLEdge components, are never deactivated, and fitted on every iteration.

Print the result of the fit

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

Visualize the result

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

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

- `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:
Fig. 69: Curve fitting quantification of a boron nitride EELS core-loss spectrum from the EELS Data Base.

- `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()`
1.14 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.

1.14.1 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
<Signal1D, 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
<Signal1D, 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 = si4D.as_signal1D(0)
>>> si4D
<Signal1D, 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
<EDSSEMSpectrum, 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
<EDSSEMSpectrum, 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:

```python
>>> 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_MnKa = 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:

```python
>>> 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:

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

or through the GUI:

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

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).

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

or through the GUI:

```python
>>> hs.preferences.gui()
```
Fig. 70: EDS microscope parameters preferences window

Fig. 71: EDS preferences window
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()
```

![Axis properties window](image)

Fig. 72: Axis properties window

Copying spectrum calibration

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

```python
>>> # s1pixel contains all the parameters
>>> s1pixel = 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)
```
1.14.2 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()
```

(continues on next page)
HyperSpy includes an elemental database, which contains the energy of the X-ray lines.

```python
text
>>> 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.
```

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.

1.14.3 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()
```

![Fig. 73: EDS spectrum](image)

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`.

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

![EDS spectrum plot with line markers](image)

*Fig. 74: EDS spectrum plot with line markers*

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'])
```

1.14.4 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 urlopen, urlretrieve
>>> from zipfile import ZipFile

>>> files = urlretrieve("https://www.dropbox.com/s/ecdlqwxjq04m5mx/"
                          ... "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):
Fig. 75: EDS spectrum plot with a selection of line markers

```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.

```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')
```

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')
```

**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. 76: Iron map as computed and displayed by `get_lines_intensity`
Fig. 77: EDS spectrum with integration windows markers

Fig. 78: EDS spectrum with background subtraction markers.
1.14.5 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 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></td>
<td></td>
</tr>
<tr>
<td>Ar_Ka</td>
<td>A</td>
<td>3136.88</td>
</tr>
<tr>
<td>Ar_Kb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_Ka</td>
<td>A</td>
<td>79258.9</td>
</tr>
<tr>
<td>Cu_Ka</td>
<td>A</td>
<td>1640.8</td>
</tr>
<tr>
<td>Cu_Kb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cu_La</td>
<td>A</td>
<td>74032.6</td>
</tr>
<tr>
<td>Cu_Lb1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cu_Ln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cu_LL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cu_Lb3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mn_Ka</td>
<td>A</td>
<td>47796.6</td>
</tr>
<tr>
<td>Mn_Kb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mn_LA</td>
<td>A</td>
<td>73665.7</td>
</tr>
<tr>
<td>Mn_Ln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mn_LL1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mn_Lb3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_La</td>
<td>A</td>
<td>68703.8</td>
</tr>
<tr>
<td>Zr_Lb1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Lb2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Ln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Lg3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Ll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Lg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zr_Lb3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>background_order_6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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:
HyperSpy Documentation, Release 1.6.0

```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)
```

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

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

Al_Ka at 1.4865 keV : Intensity = 65241.42
Ar_Ka at 2.9577 keV : Intensity = 3136.88
C_Ka at 0.2774 keV : Intensity = 79258.95
Cu_Ka at 8.0478 keV : Intensity = 1640.80
Cu_La at 0.9295 keV : Intensity = 74032.56
Mn_Ka at 5.8987 keV : Intensity = 47796.57
Mn_La at 0.63316 keV : Intensity = 73665.70
Zr_La at 2.0423 keV : Intensity = 68703.75

Finally, we visualize the result:

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

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()`
1.14.6 EDS Quantification

HyperSpy includes three methods for EDS quantification with or without absorption correction:

- 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 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()
```

(continues on next page)
>>> 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 the 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:

```python
>>> s.set_microscope_parameters(probe_area=0.00125)
```

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.

Absorption Correction

Absorption correction can be included into any of the three quantification methods by adding the parameter absorption_correction=True to the function. By default the function iterates the quantification function until of tolerance value of 0.5% up to a maximum number of iterations. The maximum number of iterations is set to 30 by default but can be increased by specifying max_iterations= in the function call. However, typically for TEM experiments convergence is witness after less than 5 iterations.

For example:

```python
>>> s.quantification(intensities, method='cross_section',
                   factors=factors, absorption_correction=True)
```

However for the kfactor method the user must additionally provide a sample thickness (in nm) either as a single float value or as a numpy array with the same dimensions as the navigation axes. If this is done the calculated mass_thickness is additionally outputted from the function as well as the composition maps for each element.

```python
>>> s.quantification(intensities, method='CL',
                   factors=factors, absorption_correction=True,
                   thickness = 100.)
```

At this stage absorption correction is only applicable for parallel-sided, thin-film samples. Absorption correction is calculated on a pixel by pixel basis after having determined a sample mass-thickness map. It therefore may be a source of error in particularly inhomogeneous specimens.

Absorption correction can also only be applied to spectra from a single EDS detector. For systems that consist of multiple detectors, such as the Thermo Fisher Super-X, it is therefore necessary to load the spectra from each detector separately.

1.14.7 Utils

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'])
a = array([[ 26330.38933818, 372.02616732]])
```

```python
>>> hs.material.mass_absorption_mixture(elements=['Al','Zn'], weight_percent=[50,50], energies='Al_Ka')
2587.4161643905127
```
Electron and X-ray range

The electron and X-ray range in a bulk material can be estimated with `hs.eds.electron_range()` and `hs.eds.xray_range()`.

To calculate the X-ray range of Cu Ka in pure Copper at 30 kV in micron:

```python
>>> hs.eds.xray_range('Cu_Ka', 30.)
1.9361716759499248
```

To calculate the X-ray range of Cu Ka in pure Carbon at 30kV in micron:

```python
>>> hs.eds.xray_range('Cu_Ka', 30., hs.material.elements.C.
>>>                     Physical_properties.density_gcm3)
7.6418811280855454
```

To calculate the electron range in pure Copper at 30 kV in micron

```python
>>> hs.eds.electron_range('Cu', 30.)
2.8766744984001607
```

1.15 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`.

1.15.1 Number of effective electrons

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

$$n_{\text{eff1}}(\epsilon^{-1}) = \frac{2\epsilon_0 m_0}{\pi^2 \epsilon^2 n_a} \int_0^E E' \sqrt{\frac{1}{\epsilon} \left(\frac{1}{\epsilon} - 1\right)} dE'$$

$$n_{\text{eff2}}(\epsilon_2) = \frac{2\epsilon_0 m_0}{\pi^2 \epsilon^2 n_a} \int_0^E E' \epsilon_2(E') dE'$$

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.
1.15.2 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:

\[ S(E) = \frac{2I_0t}{\pi\alpha_0\alpha_0^t} \ln \left[ 1 + \left( \frac{\beta}{\theta(E))} \right)^2 \right] \left[ -1 \epsilon(E) \right] \]

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.

1.16 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.

1.16.1 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:
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_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`.  

Fig. 79: Unwrapped phase image.
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
>>> out_size = int(2*sb_size.data)
>>> wave_image = im.reconstruct_phase(reference_hologram,
...     sb_position=sb_position, sb_size=sb_size,
...     output_shape=(out_size, out_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.

**Further processing of complex wave and phase**

Once the complex electron wave reconstructed it can be processed the same way as any other complex signal. A useful tool to explore the complex data is Argand plot, which can be calculated and displayed as follows:

```python
>>> ad = wave_image.argand_diagram(display_range=[-3, 3])
>>> ad.plot(scalebar=False)
```

**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)
```
Fig. 80: Argand diagram of the reconstructed complex wave.
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, the 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()`.

### 1.17 Loading and saving data

#### Contents

- Loading and saving data
  - Loading files: the load function
    - Loading multiple files
  - Saving data to files
  - Supported formats
    - HSpy - HyperSpy's HDF5 Specification
    - NetCDF
    - MRC
    - MRCZ
    - EMSA/MSA
    - Ripple
    - Images
    - TIFF
    - Gatan Digital Micrograph
    - EDAX TEAM SPD and SPC
    - FEI TIA SER and EMI
    - SEMPER UNF binary format
    - Blockfile
    - DENS heater log
    - Bruker’s formats
    - EMD
    - Protochips log
    - USID
    - Nexus
    - SUR and PRO format
1.17.1 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 loading was successful, the variable `s` contains a HyperSpy signal or a signal of the HyperSpy extensions - see available signal subclasses for more information. If the loaded file contains several datasets, the `load()` functions will return a list of the corresponding signal.

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

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.:

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

```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:

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

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

```python
>>> # Say there are two files like this:
>>> # /home/data/afile[1x1].hspy
>>> # /home/data/afile[1x2].hspy
>>> s = hs.load('/home/data/afile[\*].hspy', escape_square_brackets=True)
```

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.:

```python
>>> ls
CL1.raw  CL1.rpl  CL2.rpl  CL3.rpl  CL4.rpl  LL3.raw  shift_map-
CL1.rpl  CL2.rpl  CL3.rpl  CL4.rpl  hdf5/   LL3.rpl
```

```python
>>> s = hs.load('*.rpl')
>>>
s
[<EELSSpectrum, title: CL1, dimensions: (64, 64, 1024)>,
 <EELSSpectrum, title: CL2, dimensions: (64, 64, 1024)>,
 <EELSSpectrum, title: CL3, dimensions: (64, 64, 1024)>,
 <EELSSpectrum, title: CL4, dimensions: (64, 64, 1024)>,
 <EELSSpectrum, title: LL3, dimensions: (64, 64, 1024)>]
```

```python
>>> s = hs.load('*.rpl', stack=True)
>>>
s
<EELSSpectrum, title: mva, dimensions: (5, 64, 64, 1024)>
```

### 1.17.2 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 you want to save in the *ripple format* write instead:

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

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

### 1.17.3 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>
<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>hspx</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>No</td>
</tr>
<tr>
<td>EMSA/MSA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NetCDF</td>
<td>Yes</td>
<td>No</td>
<td>Yes</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>
<tr>
<td>h5USID .h5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Phenom .elid</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DigitalSurf’s .sur and .pro</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Nexus .nxs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EMPAD .xml</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 workarround 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']
>>> # 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, dimensions: (|)>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: C_Ka at 0.28 keV, dimensions: (|)>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Cu_La at 0.93 keV, dimensions: (|)>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Mn_La at 0.63 keV, dimensions: (|)>,
 <BaseSignal, title: X-ray line intensity of EDS SEM Signal1D: Zr_La at 2.04 keV, dimensions: (|)>]
```

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 *HSpy - HyperSpy’s HDF5 Specification* 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](https://www.ccp-em.org/). 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:

- `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](https://github.com/em-MRCZ) PyPI: [https://pypi.python.org/pypi/mrcz](https://pypi.python.org/pypi/mrcz) Citation: Submitted. Preprint: [http://www.biorxiv.org/content/early/2017/03/13/116533](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.
• \texttt{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 \texttt{do\_async = True} you may wish to leave one thread free for the Python GIL.

The recommended compression codec is ‘zstd’ (zStandard) with \texttt{clevel=1} for general use. If speed is critical, use ‘lz4’ (LZ4) with \texttt{clevel=9}. Integer data compresses more readably 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 \texttt{compressor=None}, extension \texttt{.mrc}. JSON metadata will not be recognized by other MRC-supporting software but should not cause crashes.

**Example Usage**

```python
>>> s.save('file.mrcz', do_async=True, compressor='zstd', clevel=1)

>>> new_signal = hs.load('file.mrcz')
```

### EMSA/MSA

This open standard format is widely used to exchange single spectrum data, but it does not support multidimensional data. It can be used to exchange single spectra with Gatan’s Digital Micrograph.

**Warning:** If several spectra are loaded and stacked (\texttt{hs.load('pattern', stack_signals=True}) the calibration read from the first spectrum and applied to all other spectra.

**Extra saving arguments**

For the MSA format the \texttt{format} argument is used to specify whether the energy axis should also be saved with the data. The default, ‘Y’ omits the energy axis in the file. The alternative, ‘XY’, saves a second column with the calibrated energy data. It is possible to personalise the separator with the \texttt{separator} keyword.

**Warning:** However, if a different separator is chosen the resulting file will not comply with the MSA/EMSA standard and HyperSpy and other software may not be able to read it.

The default encoding is \texttt{latin-1}. It is possible to set a different encoding using the \texttt{encoding} argument, e.g.:

```python
>>> s.save('file.msa', encoding = 'utf8')
```
Ripple

This open standard format is widely used to exchange multidimensional data. However, it only supports data of up to three dimensions. It can be used to exchange data with Bruker and Lispix. Used in combination with the ImportRPL Digital Micrograph plugin it is very useful for exporting data to Gatan’s Digital Micrograph.

The default encoding is latin-1. It is possible to set a different encoding using the encoding argument, e.g.:

```python
>>> s.save('file.rpl', encoding = 'utf8')
```

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).

Images

HyperSpy is able to read and write data too 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.

**Warning:** It has been reported that in some versions of Gatan Digital Micrograph, any binned data stores the _averages_ of the binned channels or pixels, rather than the _sum_, which would be required for proper statistical analysis. We therefore strongly recommend that all binning is performed using Hyperspy where possible. See the original bug report here.

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_fnames: [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_fnames: [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 downsample 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):

```python
>>> 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:

```python
>>> hs.load("sample80kv.bcf")
[<Signal2D, title: BSE, dimensions: (100, 75)>,
 <Signal2D, title: SE, dimensions: (100, 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.
HyperSpy Documentation, Release 1.6.0

>>> hs.load("sample80kv.bcf", cutoff_at_kV=80)
[[Signal2D, title: BSE, dimensions: (100, 75)],
 Signal2D, title: SE, dimensions: (100, 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). This format is used by the prismatic software to save the simulation outputs.

Extra loading arguments

- dataset_path: None, str or list of str. Path of the dataset. If None, load all supported datasets, otherwise the specified dataset(s).
- stack_group: bool, default is True. Stack datasets of groups with common path. Relevant for emd file version >= 0.5 where groups can be named ‘group0000’, ‘group0001’, etc.

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/data")
```

Or several by using a list:

```python
>>> s = hs.load("adatafile.emd",
... dataset_name=[
... "/experimental/science_data_1/data",
... "/experimental/science_data_2/data"])
```

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 Velox EMD files if 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.

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.

```python
>>> hs.load("sample.emd", sum_EDS_detectors=False)
[<Signal2D, title: HAADF, dimensions: (179, 161)>,
 <EDSSEMSpectrum, title: EDS - SuperXG21, dimensions: (179, 161|4096)>,
 <EDSSEMSpectrum, title: EDS - SuperXG22, dimensions: (179, 161|4096)>,
 <EDSSEMSpectrum, title: EDS - SuperXG23, dimensions: (179, 161|4096)>,
 <EDSSEMSpectrum, title: EDS - SuperXG24, dimensions: (179, 161|4096)>]
```

(continues on next page)
HyperSpy Documentation, Release 1.6.0

(continued from previous page)

```python
>>> hs.load("sample.emd", sum_frames=False, load_SI_image_stack=True, SI_dtype=np.int8, rebin_energy=4)
[<Signal2D, title: HAADF, dimensions: (50|179, 161)>,
 <EDSSEMSpectrum, title: EDS, dimensions: (50, 179, 161|1024)>]
```

Protochips log

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.

USID

Background

Universal Spectroscopy and Imaging Data (USID) is an open, community-driven, self-describing, and standardized schema for representing imaging and spectroscopy data of any size, dimensionality, precision, instrument of origin, or modality. USID data is typically stored in Hierarchical Data Format Files (HDF5) and the combination of USID within HDF5 files is referred to as h5USID.

pyUSID provides a convenient interface to I/O operations on such h5USID files. USID (via pyUSID) forms the foundation for other materials microscopy scientific python package called pycroscopy. If you have any questions regarding this module, please consider contacting the developers of pyUSID.

Requirements

1. Reading and writing h5USID files require the installation of pyUSID.
2. Files must use the .h5 file extension in order to use this io plugin. Using the .hdf5 extension will default to HyperSpy’s own plugin.

Reading

h5USID files can contain multiple USID datasets within the same file. HyperSpy supports reading in one or more USID datasets.
Extra loading arguments

- **dataset_path**: str. Absolute path of USID Main HDF5 dataset. (default is None - all USID Main Datasets will be read)
- **ignore_non_linear_dims**: bool, default is True. If True, parameters that were varied non-linearly in the desired dataset will result in Exceptions. Else, all such non-linearly varied parameters will be treated as linearly varied parameters and a Signal object will be generated.

Reading the sole dataset within a h5USID file:

```python
>>> hs.load("sample.h5")
<Signal2D, title: HAADF, dimensions: (|128, 128)>
```

If multiple datasets are present within the h5USID file and you try the same command again, all available datasets will be loaded.

**Note:** Given that HDF5 files can accommodate very large datasets, setting lazy=True is strongly recommended if the contents of the HDF5 file are not known apriori. This prevents issues with regard to loading datasets far larger than memory.

Also note that setting lazy=True leaves the file handle to the HDF5 file open. If it is important that the files be closed after reading, set lazy=False.

```python
>>> hs.load("sample.h5")
[<Signal2D, title: HAADF, dimensions: (|128, 128)>,
 <Signal1D, title: EELS, dimensions: (|64, 64, 1024)>]
```

We can load a specific dataset using the dset_path keyword argument. Setting it to the absolute path of the desired dataset will cause the single dataset to be loaded.

```python
>>> # Loading a specific dataset
>>> hs.load("sample.h5", dset_path='/Measurement_004/Channel_003/Main_Data')
<Signal2D, title: HAADF, dimensions: (|128, 128)>
```

h5USID files support the storage of HDF5 dataset with compound data types. As an (oversimplified) example, one could store a color image using a compound data type that allows each color channel to be accessed by name rather than an index. Naturally, reading in such a compound dataset into HyperSpy will result in a separate signal for each named component in the dataset:

```python
>>> hs.load("file_with_a_compound_dataset.h5")
[<Signal2D, title: red, dimensions: (|128, 128)>,
 <Signal2D, title: blue, dimensions: (|128, 128)>,
 <Signal2D, title: green, dimensions: (|128, 128)>]
```

h5USID files also support parameters or dimensions that have been varied non-linearly. This capability is important in several spectroscopy techniques where the bias is varied as a bi-polar triangular waveform rather than linearly from the minimum value to the maximum value. Since HyperSpy Signals expect linear variation of parameters / axes, such non-linear information would be lost in the axes manager. The USID plugin will default to a warning when it encounters a parameter that has been varied non-linearly:

```python
>>> hs.load("sample.h5")
UserWarning: Ignoring non-linearity of dimension: Bias
<BaseSignal, title: , dimensions: (|7, 3, 5, 2)>```

1.17. Loading and saving data
Obviously, in order to prevent accidental misinterpretation of information downstream, the keyword argument `ignore_non_linear_dims` can be set to `False` which will result in a `ValueError` instead.

```python
>>> hs.load("sample.h5")
ValueError: Cannot load provided dataset. Parameter: Bias was varied non-linearly.
Supply keyword argument "ignore_non_linear_dims=True" to ignore this error
```

**Writing**

Signals can be written to new h5USID files using the standard `save()` function. Setting the `overwrite` keyword argument to `True` will append to the specified HDF5 file. All other keyword arguments will be passed to `pyUSID.hdf_utils.write_main_dataset()

```python
>>> sig.save("USID.h5")
```

Note that the model and other secondary data artifacts linked to the signal are not written to the file but these can be implemented at a later stage.

**Nexus**

**Background**

NeXus is a common data format originally developed by the neutron, x-ray communities. It is still being developed as an international standard by scientists and programmers representing major scientific facilities in order to facilitate greater cooperation in the analysis and visualization of data. Nexus uses a variety of classes to record data, values, units and other experimental metadata associated with an experiment. For specific types of experiments an Application Definition may exist which defines an agreed common layout that facilities can adhere to. Nexus metadata and data are stored in Hierarchical Data Format Files (HDF5) with a `.nxs` extension although standards HDF5 extensions are sometimes used. Files must use the `.nxs` file extension in order to use this io plugin. Using the `.nxs` extension will default to the Nexus loader.

The loader will follow version 3 of the Nexus data rules. The signal type, Signal1D or Signal2D, will be inferred by the `interpretation` attribute, if this set to `spectrum` or `image`, in the `NXdata` description. If the `interpretation` attribute is not set the loader will return a `BaseSignal` which must then be converted to the appropriate signal type. Following the Nexus data rules if a default dataset is not defined the loader will load `NXdata` and HDF datasets according to the keyword options in the reader. A number of the Nexus examples from large facilities don’t use `NXdata` or use older versions of the Nexus implementation. Data can still be loaded from these files but information or associations may be missing. This missing information can however be recovered from within the `original_metadata` which contains the overall structure of the entry.

As the Nexus format uses HDF5 and needs to read data and metadata structured in different ways the loader is written to quite flexible and can also be used to inspect any hdf5 based file.
Differences with respect to hspy

Hyperspy metadata structure stores arrays as hdf datasets without attributes and stores floats, ints and strings as attributes. Nexus formats typically use hdf datasets attributes to store additional information such as an indication of the units for an axis or the NX_class which the dataset structure follows. The metadata, hyperspy or original_metadata, therefore needs to be able to indicate the values and attributes of a dataset. To implement this structure the `value` and `attrs` of a dataset can also be defined. The value of a dataset is set using a `value` key. The attributes of a dataset are defined by an `attrs` key.

For example to store an array, called `axis_x`, with a units attribute within original_metadata the following structure would be used:

```python
original_metadata
  axis_x
    value : array([1.0, 2.0, 3.0, 4.0, 5.0])
    attrs
      units : mm
```

```python
>>> original_metadata.set_item(axis_x.value, [1.0, 2.0, 3.0, 4.0, 5.0])
>>> original_metadata.set_item(axis_x.attrs.units, "mm")
```

To access the axis information:

```python
>>> original_metadata.axis_x.value
>>> original_metadata.axis_x.attrs.units
```

To modify the axis information:

```python
>>> original_metadata.axis_x.value = [2.0, 3.0, 4.0, 5.0, 6.0]
>>> original_metadata.axis_x.attrs.units = "um"
```

To store data in a Nexus monochromator format the `value` and `attrs` can define additional attributes.

```python
monochromator
  energy
    value : 12.0
    attrs
      units : keV
      NXclass : NXmonochromator
```

The `attrs` key can also define Nexus structures to define structures and relationships between data.

```python
mydata
  attrs
    NX_class : "NXdata"
    axes : ["x", "."]
  data
    value : [[30, 23...110]
    x
      value : [1, 2.....100]
      attrs
        unit : "mm"
```

The use of `attrs` or `value` to set values within the metadata is optional and metadata values can also be set, read or modified in the normal way.
Hyperspy metadata is stored within the Nexus file and should be automatically restored when a signal is loaded from a previously saved Nexus file.

**Note:** Altering the standard metadata structure of a signal using `attrs` or `value` keywords is not recommended.

### Reading

Nexus files can contain multiple datasets within the same file but the ordering of datasets can vary depending on the setup of an experiment or processing step when the data was collected. For example in one experiment Fe, Ca, P, Pb were collected but in the next experiment Ca, P, K, Fe, Pb were collected. HyperSpy supports reading in one or more datasets and returns a list of signals but in this example case the indexing is different. To control which data or metadata is loaded and in what order some additional loading arguments are provided.

### Extra loading arguments

- `dataset_keys`: None, str or list of strings - Default is None. Absolute path(s) or string(s) to search for in the path to find one or more datasets.
- `metadata_keys`: None, str or list of strings - Default is None. Absolute path(s) or string(s) to search for in the path to find metadata.
- `nxdata_only`: bool - Default is False. Option to only convert NXdata formatted data to signals.
- `hardlinks_only`: bool - Default is False. Option to ignore soft or External links in the file.
- `use_default`: bool - Default is False. Only load the default dataset, if defined, from the file. Otherwise load according to the other keyword options.

**Note:** Given that HDF5 files can accommodate very large datasets, setting `lazy=True` is strongly recommended if the contents of the HDF5 file are not known apriori. This prevents issues with regard to loading datasets far larger than memory.

Also note that setting `lazy=True` leaves the file handle to the HDF5 file open and it can be closed with `close_file()` or when using `compute()` with `close_file=True`.

### Reading a Nexus file a single Nexus dataset:

```python
>>> sig = hs.load("sample.nxs")
```

By default the loader will look for stored NXdata objects. If there are hdf datasets which are not stored as NXdata but which should be loaded as signals set the `nxdata_only` keyword to False and all hdf datasets will be returned as signals.

```python
>>> sig = hs.load("sample.nxs", nxdata_only=False)
```

We can load a specific datasets using the `dataset_keys` keyword argument. Setting it to the absolute path of the desired dataset will cause the single dataset to be loaded.

```python
>>> # Loading a specific dataset
>>> hs.load("sample.nxs", dataset_keys="/entry/experiment/EDS/data")
```
We can also choose to load datasets based on a search key using the `dataset_keys` keyword argument. This can also be used to load NXdata not outside of the default version 3 rules. Instead of providing an absolute path a strings to can be provided and datasets with this key will be returned. The previous example could also be written as:

```python
>>> # Loading a specific dataset
>>> hs.load("sample.nxs", dataset_keys="EDS")
```

Multiple datasets can be loaded by providing a number of keys:

```python
>>> # Loading a specific dataset
>>> hs.load("sample.nxs", dataset_keys=["EDS", "Fe", "Ca"])
```

Metadata can also be filtered in the same way using `metadata_keys`:

```python
>>> # Load data with metadata matching metadata_keys
>>> hs.load("sample.nxs", metadata_keys="entry/instrument")
```

**Note:** The Nexus loader removes any NXdata blocks from the metadata.

Nexus files also support parameters or dimensions that have been varied non-linearly. Since HyperSpy Signals expect linear variation of parameters / axes, such non-linear information would be lost in the axes manager and replaced with indices. Nexus and HDF can result in large metadata structures with large datasets within the loaded original_metadata. If lazy loading is used this may not be a concern but care must be taken when saving the data. To control whether large datasets are loaded or saved use the `metadata_keys` to load only the most relevant information.

**Writing**

Signals can be written to new Nexus files using the standard `save()` function.

**Extra saving arguments**

- `save_original_metadata`: bool - Default is True, Option to save the original_metadata when storing to file.
- `use_default`: bool - Default is False. Set the `default` attribute for the Nexus file.

```python
>>> sig.save("output.nxs")
```

Using the save method will store the nexus file with the following structure:

```
entry1
  | signal_name
  | auxiliary
  |  original_metadata
  |  hyperspy_metadata
  |  learning_results
  |  signal_data
  |  data and axes (NXdata format)
```

The original_metadata can include hdf datasets which you may not wish to store. The original_metadata can be omitted using `save_original_metadata`.
To save multiple signals the `file_writer` method can be called directly.

```python
>>> from hyperspy.io_plugins.nexus import file_writer
>>> file_writer("test.nxs", [signal1, signal2])
```

When saving multiple signals a default signal can be defined. This can be used when storing associated data or processing steps along with a final result. All signals can be saved but a single signal can be marked as the default for easier loading in hyperspy or plotting with Nexus tools. The default signal is selected as the first signal in the list.

```python
>>> from hyperspy.io_plugins.nexus import file_writer
>>> import hyperspy.api as hs

>>> file_writer("test.nxs", [signal1, signal2], use_default = True)

>>> hs.load("test.nxs", use_default = True)
```

The output will be arranged by signal name.

```
entry1 (NXentry)
  signal_name (NXentry)
    auxiliary (NXentry)
      original_metadata (NXcollection)
      hyperspy_metadata (NXcollection)
      learning_results (NXcollection)
    signal_data (NXdata format)
      data and axes
entry2 (NXentry)
  signal_name (NXentry)
    auxiliary (NXentry)
      original_metadata (NXcollection)
      hyperspy_metadata (NXcollection)
      learning_results (NXcollection)
    signal_data (NXdata)
      data and axes
```

**Note:** Signals saved as nxs by this plugin can be loaded normally and the original_metadata, signal data, axes, metadata and learning_results will be restored. Model information is not currently stored. Nexus does not store how the data should be displayed. To preserve the signal details an additional navigation attribute is added to each axis to indicate if is a navigation axis.

**Inspecting**

Looking in a Nexus or HDF file for specific metadata is often useful - e.g to find what position a specific stage was at. The methods `read_metadata_from_file` and `list_datasets_in_file` can be used to load the file contents or list the hdf datasets contained in a file. The inspection methods use the same `metadata_keys` or `dataset_keys` as when loading. For example to search for metadata in a file:

```python
>>> from hyperspy.io_plugins.nexus import read_metadata_from_file
>>> read_metadata_from_file("sample.hdf5", metadata_keys=['"stagel_z"'])
{'entry': {'instrument': {'scannables': {'stage1': {'stagel_z': {'value': -9.8710000000002, 'attrs': {'gda_field_name': 'stagel_z', 'local_name': 'stagel.stagel_z',}}}}}}}
```

(continues on next page)
To list the datasets stored in the file:

```python
>>> from hyperspy.io_plugins.nexus import read_datasets_from_file
>>> list_datasets_in_file("sample.nxs")
NXdata found
 entry/xsp3_addetector
 entry/xsp3_addetector_total
 HDF datasets found
 /entry/solstice_scan/keys/uniqueKeys
 /entry/solstice_scan/scan_shape
Out[3]:
(["/entry/xsp3_addetector", "/entry/xsp3_addetector_total"],
 ["/entry/solstice_scan/keys/uniqueKeys", "/entry/solstice_scan/scan_shape"])
```

### SUR and PRO format

This is a format developed by the digitalsurf company to handle various types of scientific measurements data such as profilometer, SEM, AFM, RGB(A) images, multilayer surfaces and profiles. Even though it is essentially a surfaces format, 1D signals are supported for spectra and spectral maps. Metadata parsing is supported, including user-customised metadata, as well as the loading of files containing multiple objects packed together.

The plugin was developed based on the MountainsMap software documentation which contains a description of the binary format.

### EMPAD format

This is the file format used by the Electron Microscope Pixel Array Detector (EMPAD). It is used to store a series of diffraction patterns from scanning transmission electron diffraction measurements, with a limited set of metadata. Similarly, to the ripple format, the raw data and metadata are saved in two different files and for the EMPAD reader, these are saved in the raw and xml files, respectively. To read EMPAD data, use the xml file:

```python
>>> sig = hs.load("file.xml")
```

which will automatically read the raw data from the raw file too. The filename of the raw file is defined in the xml file, which implies changing the file name of the raw file will break reading the file.

### Phenom ELID format

This is the file format used by the software package Element Identification for the Thermo Fisher Scientific Phenom desktop SEM. It is a proprietary binary format which can contain images, single EDS spectra, 1D line scan EDS spectra and 2D EDS spectrum maps. The reader will convert all signals and its metadata into hyperspy signals.

The current implementation supports ELID files created with Element Identification version 3.8.0 and later. You can convert older ELID files by loading the file into a recent Element Identification release and then save the ELID file into the newer file format.
1.17.4 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.

1.18 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.

1.18.1 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
```

(continues on next page)
HyperSpy Documentation, Release 1.6.0

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)
...    ...
>>> nav_axis.events.index_changed.connect(on_index_changed2, ["index"])
>>> s.axes_manager.indices = (0,)
on_index_changed_called
('Axis name: ', 'x')
('Index: ', 0)
on_index_changed2_called
('Index: ', 0)
>>> def on_index_changed3():
...    print("on_index_changed3_called")
...    ...
>>> nav_axis.events.index_changed.connect(on_index_changed3, [])
>>> s.axes_manager.indices = (1,)
on_index_changed_called
('Axis name: ', 'x')
('Index: ', 1)
on_index_changed2_called
('Index: ', 1)
on_index_changed3_called

It is also possible to map trigger keyword arguments to connected function keyword arguments as follows:

```python
>>> def on_index_changed4(arg):
...    print("on_index_changed4_called")
...    print("Index: ", arg)
...    ...
>>> nav_axis.events.index_changed.connect(on_index_changed4,
...                                          {"index": "arg"})
>>> s.axes_manager.indices = (4,)
on_index_changed_called
('Axis name: ', 'x')
('Index: ', 4)
on_index_changed2_called
('Index: ', 4)
on_index_changed3_called
on_index_changed4_called
('Index: ', 4)
```
1.18.2 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)
>>> with nav_axis.events.index_changed.suppress():
...      s.axes_manager.indices = (6,)
...      ...
>>> with nav_axis.events.suppress():
...      s.axes_manager.indices = (5,)
...      ...
```

1.18.3 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)
```

1.19 Working with big data

**Warning:** All the features described in this chapter are in beta state.

Although most of them work as described, their operation may not always be optimal, well-documented and/or consistent with their in-memory counterparts.

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.
1.19.1 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)>  
```

Loading the dataset in the original unsigned integer format would require around 35GB of memory. To store it in a floating-point format one would need almost 280GB of memory. However, with the lazy processing both of these steps are near-instantaneous and require very little computational resources.

New in version 1.4: close_file()

Currently when loading an hdf5 file lazily the file remains open at least while the signal exists. In order to close it explicitly, use the close_file() method. Alternatively, you could close it on calling compute() by passing the keyword argument close_file=True e.g.:

```python
>>> s = hs.load("file.hspy", lazy=True)
>>> ssum = s.sum(axis=0)
>>> ssum.compute(close_file=True)  # closes the file.hspy file
```
Lazy stacking

Occasionally the full dataset consists of many smaller files. To combine them into a one large LazySignal, we can stack them lazily (both when loading or afterwards):

```python
>>> 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)
```

Casting signals as lazy

To convert a regular HyperSpy signal to a lazy one such that any future operations are only performed lazily, use the `as_lazy()` method:

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

1.19.2 Machine learning

Decomposition algorithms for machine learning often perform large matrix manipulations, requiring significantly more memory than the data size. To perform decomposition operation lazily, HyperSpy provides access to several “online” algorithms as well as dask’s lazy SVD algorithm. Online algorithms perform the decomposition by operating serially on chunks of data, enabling the lazy decomposition of large datasets. In line with the standard HyperSpy signals, lazy `decomposition()` offers the following online algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SVD” (default)</td>
<td>dask.array.linalg.svd()</td>
</tr>
<tr>
<td>“PCA”</td>
<td>sklearn.decomposition.IncrementalPCA</td>
</tr>
<tr>
<td>“ORPCA”</td>
<td>ORPCA</td>
</tr>
<tr>
<td>“ORNMF”</td>
<td>ORNMF</td>
</tr>
</tbody>
</table>

See also:

`decomposition()` for more details on decomposition with non-lazy signals.
1.19.3 Practical tips

Despite the limitations detailed below, most HyperSpy operations can be performed lazily. Important points 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 `rechunk` or `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 `compute()` method:

```python
>>> s
<LazySignal2D, title: , dimensions: (|512, 512)>
>>> s.compute()
[########################################] | 100% Completed | 0.1s
>>> s
<Signal2D, title: , dimensions: (|512, 512)>
```

**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: (200, 200|512, 512)>
>>> # for fastest results, just pick one signal space pixel
>>> nav = s.transpose(optimize=True).inav[256, 256]
>>> # Alternatively, sum as per default behaviour
>>> nav = s.sum(s.axes_manager.signal_axes).T
>>> nav
<LazySignal2D, title: , dimensions: (|200, 200)>
>>> # Compute the result
>>> nav.compute()
[########################################] | 100% Completed | 13.1s
>>> s.plot(navigator=nav)
```

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 `shift1D()` is not lazy.

### 1.19.4 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 `LazySignal` is the inability to modify existing data (immutability). This is a logical consequence of the DAG (tree structure, explained in *Behind the scenes – technical details*), where a complete history of the processing has to be stored to traverse later.

In fact, lazy evaluation removes the need for such operation, since only additional tree branches are added, requiring very little resources. In practical terms the following fails with lazy signals:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s += 1
Traceback (most recent call last):
  File "<ipython-input-6-1bd1db4187be>", line 1, in <module>
    s += 1
  File "<string>", line 2, in __iadd__
  File "/home/fjd29/Python/hyperspy3/hyperspy/signal.py", line 1591, in _binary_­operator_ruler
    getattr(self.data, op_name)(other)
AttributeError: 'Array' object has no attribute '__iadd__'
```

However, when operating lazily there is no clear benefit to using in-place operations. So, the operation above could be rewritten as follows:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> s = s + 1
```

Or even better:

```
>>> s = hs.signals.BaseSignal([0]).as_lazy()
>>> sl = s + 1
```
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.

1.19.5 Behind the scenes –technical details

Standard HyperSpy signals load the data into memory for fast access and processing. While this behaviour gives good performance in terms of speed, it obviously requires at least as much computer memory as the dataset, and often twice that to store the results of subsequent computations. This can become a significant problem when processing very large datasets on consumer-oriented hardware.

HyperSpy offers a solution for this problem by including `LazySignal` and its derivatives. The main idea of these classes is to perform any operation (as the name suggests) lazily (delaying the execution until the result is requested (e.g. saved, plotted)) and in a blocked fashion. This is achieved by building a “history tree” (formally called a Directed Acyclic Graph (DAG)) of the computations, where the original data is at the root, and any further operations branch from it. Only when a certain branch result is requested, the way to the root is found and evaluated in the correct sequence on the correct blocks.

The “magic” is performed by (for the sake of simplicity) storing the data not as `numpy.ndarray`, but `dask.array.Array` (see the `dask` documentation). `dask` offers a couple of advantages:

- **Arbitrary-sized data processing is possible.** By only loading a couple of chunks at a time, theoretically any signal can be processed, albeit slower. In practice, this may be limited: (i) some operations may require certain chunking pattern, which may still saturate memory; (ii) many chunks should fit into the computer memory comfortably at the same time.

- **Loading only the required data.** If a certain part (chunk) of the data is not required for the final result, it will not be loaded at all, saving time and resources.

- **Able to extend to a distributed computing environment (clusters).** :py:`dask.distributed` (see the `dask` documentation) offers a straightforward way to expand the effective memory for computations to that of a cluster, which allows performing the operations significantly faster than on a single machine.

1.20 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.
<table>
<thead>
<tr>
<th>Energy Resolution MnKa (eV)</th>
<th>Live Time (s)</th>
<th>Real Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beam Current (nA)</td>
<td>Beam Energy (keV)</td>
<td></td>
</tr>
<tr>
<td>Probe Area (nm²)</td>
<td>Convergence Angle (mrad)</td>
<td></td>
</tr>
<tr>
<td>Magnification</td>
<td>Microscope</td>
<td></td>
</tr>
<tr>
<td>Stage</td>
<td>Rotation (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tilt Alpha (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tilt Beta (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X (mm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y (mm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z (mm)</td>
<td></td>
</tr>
<tr>
<td>Working Distance (mm)</td>
<td>TEM Detector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDS Azimuth Angle (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Elevation Angle (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy Resolution MnKa (eV)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Live Time (s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Real Time (s)</td>
<td></td>
</tr>
<tr>
<td>EELS</td>
<td>Aperture (mm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Collection Angle (mrad)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dwell Time (s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exposure (s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Frame Number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Spectrometer</td>
<td></td>
</tr>
<tr>
<td>Biprism</td>
<td>Azimuth Angle (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Position</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Voltage (V)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acquisition Mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Beam Current (nA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Beam Energy (keV)</td>
<td></td>
</tr>
<tr>
<td>Camera Length (mm)</td>
<td>Convergence Angle (mrad)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Magnification</td>
<td></td>
</tr>
<tr>
<td>Stage</td>
<td>Rotation (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tilt Alpha (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tilt Beta (°)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X (mm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y (mm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z (mm)</td>
<td></td>
</tr>
</tbody>
</table>

**General**
- Authors
- Date
- DOI
- Original Filename
- Notes
1.20.1 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.
1.20.2 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 nm².

**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 nm².

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.

elevation_angle type: Float

The elevation angle of the detector in degree. The detector is perpendicular to the surface with an angle of 90.

energy_resolution_MnKa type: Float

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.

real_time type: Float

The time spent to record the spectrum in second.

live_time type: Float

The time spent to record the spectrum in second, compensated for the dead time of the detector.

Biprism

This node stores parameters of biprism used in off-axis electron holography

azimuth_angle (*) type: Float

Rotation angle of the biprism in degree

position type: Str

Position of the biprism in microscope column, e.g. Selected area aperture plane

tVoltage type: Float

Voltage of electrostatic biprism in volts
1.20.3 Sample

credits type: Str
Acknowledgment of sample supplier, e.g. Prepared by Putin, Vladimir V.

description type: Str
A brief description of the sample

elements type: list
A list of the symbols of the elements composing the sample, e.g. ['B', 'N'] for a sample composed of Boron and Nitrogen.

xray_lines type: list
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.

thickness type: Float
The thickness of the sample in m.

1.20.4 Signal

signal_type type: Str
A term that describes the signal type, e.g. EDS, PES... This information can be used by HyperSpy to load the file as a specific signal class and therefore the naming should be standarised. 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

1.20.5 _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.

1.21 Bibliography

1.21.1 Bibliography


1.21.2 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 a user, but, just in case: the best way to start understanding how HyperSpy works and to build a broad overview of the code as it stands is to use it – so what are you waiting for? Install HyperSpy.

The HyperSpy User-Guide also provides a good overview of all the parts of the code that are currently implemented as well as much information about how everything works – so read it well.

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 will be awesome if you could help others with their questions.

2. Issues reported in the issues tracker are precious contributions.

3. Pull request reviews are essential for the sustainability of open development software projects and HyperSpy is no exception. Therefore, reviews are highly appreciated. While you may need a good familiarity with the HyperSpy code base to review complex contributions, you can start by reviewing simpler ones such as documentation contributions or simple bug fixes.

4. Last but not least, you can contribute code in the form of documentation, bug fixes, enhancements or new features. That is the main topic of the rest of this guide.

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 partially finished projects just sitting there waiting for a keen new contributor to come and learn by finishing.

The documentation (let it be the docstrings, guides or the website) is always in need of some care. Besides, contributing to HyperSpy’s documentation is a very good way to get familiar with GitHub.

When you’ve decided what you’re going to work on – let people know using the online forums! It may be that someone else is doing something similar and can help.; it is also good to make sure that those working on related projects are pulling in the same direction.

There are 3 key points to get right when starting out as a contributor:

1. Work out what you want to contribute and break it down in to manageable chunks. Use Git branches to keep work separated in manageable sections.

2. Make sure that your code style is good.

3. Bear in mind that every new function you write will need tests and user documentation!

2.2 Using Git and GitHub

For developing the code, the home of HyperSpy is on GitHub, and you’ll see that a lot of this guide boils down to properly use that platform. So, visit the following link and poke around the code, issues, and pull requests: HyperSpy on GitHub.

It is probably also worth visiting the github.com and to go through the “boot camp” to get a feel for the terminology.

In brief, to give you a hint on the terminology to search for and get accustomed to, the contribution pattern is:

1. Setup git/github, if you don’t have it yet.

2. Fork HyperSpy on GitHub.

3. Checkout your fork on your local machine.

4. Create a new branch locally, where you will make your changes.

5. Push the local changes to your own HyperSpy fork on GitHub.
6. Create a pull request (PR) to the official HyperSpy repository.

**Note:** You cannot mess up the main HyperSpy project unless you have been promoted to write access and the dev-team. So when you’re starting out be confident to play, get it wrong, and if it all goes wrong, you can always get a fresh install of HyperSpy!!

PS: If you choose to develop in Windows/Mac you may find Github Desktop useful.

### 2.2.1 Use Git and work in manageable branches

By now you will have had a look around GitHub – but why is it so important?

Well, GitHub is the public forum in which we manage and discuss development of the code. More importantly, it enables every developer to use *Git*, which is an open source “version control” system. By version control, we mean that you can separate out your contribution to the code into many versions (called branches) and switch between them easily. Later, you can choose which version you want to have integrated into HyperSpy. You can learn all about Git at [git-scm](https://git-scm.com)!

It is very important to separate your contributions so that each branch is a small advancement on the “master” code or on another branch. In the end, each branch will have to be checked and reviewed by someone else before it can be included – so if it is too big, you will be asked to split it up!

For personal use, before integrating things into the main HyperSpy code, you can merge some together for your personal use. However, make sure each new feature has its own branch that is contributed through a separate pull request!

Diagrammatically, you should be aiming for something like this:
2.3 Running and writing tests

2.3.1 Writing tests

Every new function that is written into HyperSpy needs to be tested and documented. HyperSpy uses the pytest library for testing. The tests reside in the hyperspy.tests module.

Tests are short functions, found in hyperspy/tests, that call your functions under some known conditions and check the outputs against known values. They should depend on as few other features as possible so that when they break we know exactly what caused it. Ideally, the tests should be written at the same time as the code itself, as they are very convenient to run to check outputs when coding. Writing tests can seem laborious but you’ll probably soon find that they’re very important as they force you to sanity check all you do.

Useful hints on testing:

• When comparing integers, it’s fine to use ==
• When comparing floats, be sure to use np.testing.assert_almost_equal() or np.testing.assert_allclose()
• np.testing.assert_allclose() is also convenient for comparing numpy arrays
• The hyperspy.misc.test_utils.py contains a few useful functions for testing
• @pytest.mark.parametrize() is a very convenient decorator to test several parameters of the same function without having to write to 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 object, for example:

```python
>>> del s
>>> gc.collect()
```

2.3.2 Running tests

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
pip install -e .[test]

# Or just installing the dependencies using conda
conda install -c conda-forge pytest pytest-mpl
```

To run them:

```
pytest --mpl --pyargs hyperspy
```

Or, from HyperSpy’s project folder, simply:

```
pytest
```
Note: pytest configuration options are set in the setup.cfg file, under the [tool:pytest] section. See the pytest configuration documentation for more details.

2.3.3 Test coverage

Once, you have pushed your pull request to the official HyperSpy repository, it can be useful to check the coverage of your tests using the codecov.io check of your PR. There should be a link to it at the bottom of your PR on the Github PR page. This service can help you to find how well your code is being tested and exactly which parts are not currently tested.

You can also measure code coverage locally. If you have installed pytest-cov, you can run (from HyperSpy’s project folder):

```
pytest --cov=hyperspy
```

Configuration options for code coverage are also set in the setup.cfg file, under the [coverage:run] and [coverage:report] sections. See the coverage documentation for more details.

2.3.4 Continuous integration (CI)

The test suite is run using continuous integration services provided by Travis CI and Azure Pipeline. The CI helper scripts are pulled from the ci-scripts repository.

The testing matrix is as follow:

- **Travis CI**: test all supported python versions on Linux; all dependencies are pulled from pypi,
- **Azure Pipeline**: test a range of python version on Linux, MacOS and Windows; all dependencies are pulled from anaconda cloud using the Anaconda “defaults” and the “conda-forge” channel (in this order of priority)

This testing matrix has been designed to be simple and easy to maintain and also to ensure that packages from pypi and Anaconda cloud are not mixed in order to avoid red herring failures of the test suite caused by application binary interface (ABI) incompatibility between dependencies.

The most recent versions of packages will be available first on pypi and later on anaconda cloud. It means that if a recent release of a dependency breaks the test suite, it should happen first on travis - usual suspect would be matplotlib, numpy, scipy, etc. Similarly, deprecation warning should appear first on Travis CI.

The build of the doc is done on Travis CI and it is worth checking that no new warnings have been introduced when writing documentation in the user guide or in the docstring.

2.3.5 Plot testing

Plotting is tested using the @pytest.mark.mpl_image_compare decorator of the pytest mpl plugin. This decorator uses reference images to compare with the generated output during the tests. The reference images are located in the folder defined by the argument baseline_dir of the @pytest.mark.mpl_image_compare decorator.

To run plot tests, you simply need to add the option --mpl:

```
pytest --mpl
```
If you don’t use --mpl, the 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 decorators such as @pytest.mark.mpl_image_compare.
2. If you need to generate a new reference image in the folder plot_test_dir, for example, run: pytest --mpl-generate-path=plot_test_dir
3. Run again the tests and this time they should pass.
4. Use git add to put the new file in the git repository.

When the plotting tests are failing, it is possible to download the figure comparison images generated by pytest-mpl in the artifacts tabs of the corresponding build on azure pipeline:

The plotting tests are tested on azure pipeline against a specific version of matplotlib defined in conda_environment_dev.yml since small changes in the way matplotlib generates the figure can make the tests fail.

For plotting tests, the matplotlib backend is set to agg by setting the MPLBACKEND environment variable to agg. At the first import of matplotlib.pyplot, matplotlib will look at the MPLBACKEND environment variable and accordingly set the backend.

### 2.3.6 Exporting pytest results as HTML

With pytest-html it is possible to export the results of running pytest for easier viewing. It can be installed by conda:

```
conda install pytest-html
```

and run by:

```
pytest --mpl --html=report.html
```

See pytest-mpl for more details.
2.4 Writing documentation

Documentation comes in two parts: docstrings and user-guide documentation.

2.4.1 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.

You can check your docstrings follow the convention by using the flake8-docstrings extension, like this:

```
# If not already installed, you need flake8 and flake8-docstrings
pip install flake8 flake8-docstrings

# Run flake8 on your file
flake8 /path/to/your/file.py

# Example output
/path/to/your/file.py:46:1: D103 Missing docstring in public function
/path/to/your/file.py:59:1: D205 1 blank line required between summary line and __doc__
```

2.4.2 User-guide documentation

A description of the functionality of the code and how to use it with examples and links to the relevant code. When writing both the docstrings and user guide documentation, it is useful to have some data which the users can use themselves. Artificial datasets for this purpose can be found in hyperspy.datasets.artificial_data.

2.4.3 Build the documentation

To check the output of what you wrote, you can build the documentation by running the make command in the hyperspy/doc directory. For example make html will build the whole documentation in html format. See the make command documentation for more details.

To install the documentation dependencies, run either

```
$ conda install hyperspy-dev
```

or

```
$ pip install hyperspy[build-doc]
```

When writing documentation, the Python package sphobjinv can be useful for writing cross-references. For example, to find how to write a cross-reference to hyperspy.signal.BaseSignal.set_signal_type(), use:

```
$ sphobjinv suggest doc/_build/html/objects.inv set_signal_type -st 90
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
</table>
2.5 Coding style

HyperSpy follows the Style Guide for Python Code - these are rules for code consistency that you can read all about in the Python Style Guide. You can use the black code formatter to automatically fix the style of your code. You can install and run black by:

```
pip install black
black /path/to/your/file.py
```

In Linux and MacOS you can run black automatically after each commit by adding a post-commit file to .git/hook with the following content:

```
#!/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
    # black correction
    black -v $f
    git add $f
  done
#git commit -m "Automatic style corrections courtesy of black"
GIT_COMMITTER_NAME="black" GIT_COMMITTER_EMAIL="black@email.com" git commit --author="black <black@email.com>" -m "Automatic style corrections courtesy of black"
```

2.6 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](https://www.hyperspy.org). While interfaces of the two arrays are indeed almost identical, the most important differences are (da being dask.array.Array in the examples):

- **Dask arrays are immutable**: da[3] = 2 does not work. da += 2 does, but it’s actually a new object – you might as well use da = da + 2 for a better distinction.

- **Unknown shapes are problematic**: res = da[da>0.3] works, but the shape of the result depends on the values and cannot be inferred without execution. Hence, few operations can be run on res lazily, and it should be avoided if possible.

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 map, separate cases for lazy signals will have to be written. If a function operates on arbitrary-sized arrays and the shape of the output can be known before calling, da.map_blocks and da.map_overlap are efficient and flexible.

Finally, in addition to _iterate_signal that is available to all HyperSpy signals, lazy counterparts also have the _block_iterator method that supports signal and navigation masking and yields (returns on subsequent calls) the
underlying dask blocks as numpy arrays. It is important to note that stacking all (flat) blocks and reshaping the result into the initial data shape will not result in identical arrays. For illustration it is best to see the dask documentation.

2.7 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.

2.7.1 Writing Numba code

If you need to improve the speed of a given part of the code your first choice should be Numba. The motivation is that Numba code is very similar (when not identical) to Python code, and, therefore, it is a lot easier to maintain than Cython code (see below).

Numba is also a required dependency for HyperSpy, unlike Cython which is only an optional dependency.

2.7.2 Writing Cython code

Cython code should only be considered if:

1. It is not possible to speed up the function using Numba, and instead,

2. it is accompanied by a pure Python version of the same code that behaves exactly in the same way when the compiled C extension is not present. This extra version is required because we may not be able to provide binaries for all platforms and not all users will be able to compile C code in their platforms.

Please read through the official Cython recommendations (http://docs.cython.org/) before writing Cython code.

To help troubleshoot potential deprecations in future Cython releases, add a comment in the header of your .pyx files stating the Cython version you used when writing the code.

Note that the “cythonized” .c or .cpp files are not welcome in the git source repository because they are typically very large.

Once you have written you Cython files, add them to raw_extensions in setup.py.

Compiling Cython code

If Cython is present in the build environment and any cythonized c/c++ file is missing, then setup.py tries to cythonize all extensions automatically.

To make the development easier setup.py provides a recythonize command that can be used in conjunction with default commands. For example

```
python setup.py recythonize build_ext --inplace
```

will recythonize all Cython code and compile it.

Cythonization and compilation also take place during continuous integration (CI).
2.8 Writing packages that extend HyperSpy

New in version 1.5: External packages can extend HyperSpy by registering signals, components and widgets.

**Warning:** The mechanism to register extensions is in beta state. This means that it can change between minor and patch versions. Therefore, if you maintain a package that registers HyperSpy extensions, please verify that it works properly with any future HyperSpy release. We expect it to reach maturity with the release of HyperSpy 2.0.

External packages can extend HyperSpy by registering signals, components and widgets. Objects registered by external packages are “first-class citizens” i.e. they can be used, saved and loaded like any of those objects shipped with HyperSpy. Because of HyperSpy’s structure, we anticipate that most packages registering HyperSpy extensions will provide support for specific sorts of data.

Models can also be provided by external packages, but don’t need to be registered. Instead, they are returned by the `create_model` method of the relevant `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 of 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 open software ecosystem.

2.8.1 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.

2.8.2 Creating new HyperSpy BaseSignal subclasses

When and where to 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.
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:

```yaml
signals:
  MySignal:
    signal_type: "MySignal"
    signal_type_aliases:
      - MS
      - ThisIsMySignal
    # The dimension of the signal subspace. For example, 2 for images, 1 for spectra. If the signal can take any signal dimension, set it to -1.
    signal_dimension: 1
    # The data type, "real" or "complex".
    dtype: real
    # True for LazySignal subclasses
    lazy: False
    # The module where the signal is located.
    module: my_package.signal
```

Note that HyperSpy uses `signal_type` to determine which class is the most appropriate to deal with a particular sort of data. Therefore, the signal type must be specific enough for HyperSpy to find a single signal subclass match for each sort of data.

**Warning:** HyperSpy assumes that only one signal subclass exists for a particular `signal_type`. It is up to external package developers to avoid `signal_type` clashes, typically by collaborating in developing a single package per data type.

The optional `signal_type_aliases` are used to determine the most appropriate signal subclass when using `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`.

---

2.8. Writing packages that extend HyperSpy  233
2.8.3 Creating new HyperSpy model components

When and where to create a new component

HyperSpy provides the `hyperspy._components.expression.Expression` component that enables easy creation of 1D and 2D components from mathematical expressions. Therefore, strictly speaking, we only need to create new components when they cannot be expressed as simple mathematical equations. However, HyperSpy is all about simplifying the interactive data processing workflow. Therefore, we consider that functions that are commonly used for model fitting, in general or specific domains, are worth adding to HyperSpy itself (if they are of common interest) or to specialized external packages extending HyperSpy.

The flowchart below can help you decide when and where to add a new hyperspy model `hyperspy.component`. Component for your function, should you consider creating your own.

![Flowchart](image)

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 `hyperpy_extension.yaml` file as in the following example:

```yaml
components1D:
  # _id_name of the component. It must be a UUID4. This can be generated
  # using `uuid.uuid4()`. Also, many editors can automatically generate
  # UUIDs. The same UUID must be stored in the components `_id_name` attribute.
  fc731a2c-0a05-4acb-91df-d15743b531c3:
    # The module where the component class is located.
    module: my_package.components
    # The actual class of the component
    class: MyComponent1DClass
```

Equivalently, to add a new component 2D:
components2D:

```
#: _id_name of the component. It must be a UUID4. This can be generated
#: using `uuid.uuid4()`. Also, many editors can automatically generate
#: UUIDs. The same UUID must be stored in the components `__id_name__` attribute.
#: 2ffbe0b5-a991-4fc5-a089-d2818a80a7e0:
#:   #: The module where the component is located.
#: module: my_package.components
#: class: MyComponent2DClass
```

Note: HyperSpy’s legacy components use their class name instead of a UUID as `__id_name__`. This is for compatibility with old versions of the software. New components (including those provided through the extension mechanism) must use a UUID4 in order to i) avoid name clashes ii) make it easy to find the component online if e.g. the package is renamed or the component relocated.

### 2.8.4 Creating and registering new widgets and toolkeys

To generate GUIs of specific methods and functions, HyperSpy use widgets and toolkeys:

- **widgets** (typically ipywidgets or traitsui objects) generate GUIs,

- **toolkeys** are functions using 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:

```
GUI:
#: In order to assign a widget to a function, that function must declare
#: a `toolkey`. The `toolkeys` list contains a list of all the toolkeys
#: provided by extensions. In order to avoid name clashes, by convention,
#: toolkeys must start with the name of the package that provides them.
#: toolkeys:
#:   - my_package.MyComponent
```

2.8. Writing packages that extend HyperSpy
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.

```python
GUI:
    widgets:
        ipywidgets:
            # Each widget is declared using a dictionary with two keys, 'module' and 'function'.
            my_package.MyComponent:
                # The function that creates the widget
                function: get_mycomponent_widget
                # The module where the function resides.
                module: my_package.widgets
```

2.9 Useful information

2.9.1 NEP 29 — Recommend Python and Numpy version support

Abstract

NEP 29 (NumPy Enhancement Proposals) recommends that all projects across the Scientific Python ecosystem adopt a common “time window-based” policy for support of Python and NumPy versions. Standardizing a recommendation for project support of minimum Python and NumPy versions will improve downstream project planning.

Implementation recommendation

This project supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of `numpy` released in the 24 months prior to the project, and at minimum the last three minor versions.

In `setup.py`, the `python_requires` variable should be set to the minimum supported version of Python. All supported minor versions of Python should be in the test matrix and have binary artifacts built for the release.

Minimum Python and NumPy version support should be adjusted upward on every major and minor release, but never on a patch release.

2.9.2 Conda-forge packaging

The feedstock for the conda package lives in the conda-forge organisation on github: `conda-forge/hyperspy-feedstock`. 
3.1 hyperspy package

3.1.1 Subpackages

hyperspy._components package

Submodules

hyperspy._components.arctan module

```python
class hyperspy._components.arctan.Arctan(A=1.0, k=1.0, x0=1.0, module=['numpy', 'scipy'], **kwargs):
    Bases: hyperspy._components.expression.Expression

Arctan function component.

\[ f(x) = A \cdot \arctan(k(x - x_0)) \]
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>x0</td>
<td>x0</td>
</tr>
</tbody>
</table>

Parameters

- **A** (*float*) – Amplitude parameter. \( \lim_{x \to -\infty} f(x) = -A \) and \( \lim_{x \to \infty} f(x) = A \)
- **k** (*float*) – Slope (steepness of the step). The larger \( k \), the sharper the step.
- **x0** (*float*) – Center parameter (position of zero crossing \( f(x_0) = 0 \)).

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (*str*) – Component function in SymPy text expression format with substitutions separated by \\( \cdot \). 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", "scipy"), 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.

**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>)
```
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.

\[ f(x) = \left( \frac{a + b \cdot x}{c} \right)^{-1/c} \]

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.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (str) – Component function in SymPy text expression format with substitutions separated by ".". See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \( x \) and \( y \) must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** (str) – Name of the component.
- **position** (str, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. ("x0", "y0").
- **module** ("numpy", "numexpr", "scipy", 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.

**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>)
```

#### grad_a(x)

Returns \( \frac{d(function)}{d(parameter_1)} \)

#### grad_b(x)

Returns \( \frac{d(function)}{d(parameter_1)} \)

#### grad_c(x)

Returns \( \frac{d(function)}{d(parameter_1)} \)

### hyperspy_components.doniach module

**class hyperspy._components.doniach.Doniach**

**Bases:** hyperspy._components.expression.Expression

Doniach Sunjic lineshape

\[
f(x) = \frac{A \cos \left[ \frac{\pi x}{2} + (1 - \alpha) \tan^{-1} \left( \frac{x - centre + dx}{\sigma} \right) \right]}{\left( \sigma^2 + (x - centre + dx)^2 \right)^{\frac{1 - \alpha}{2}}}
\]

\[
dx = \frac{2.354820 \sigma}{2 \tan \left[ \frac{x - centre}{2 - \alpha} \right]}
\]
Parameters

- **A** (*float*) – Height
- **sigma** (*float*) – Variance parameter of the distribution
- **alpha** (*float*) – Tail or asymmetry parameter
- **centre** (*float*) – Location of the maximum (peak position).
- ****kwargs – Extra keyword arguments are passed to the Expression component.

Note: This is an asymmetric lineshape, originally design for xps but generally useful for fitting peaks with low side tails See Doniach S. and Sunjic M., J. Phys. 4C31, 285 (1970) or http://www.casaxps.com/help_manual/line_shapes.htm for a more detailed description

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (*str*) – Component function in SymPy text expression format with substitutions separated by . See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** (*str*) – Name of the component.
- **position** (*str, optional*) – The parameter name that defines the position of the component if applicable. It enables iterative 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", "scipy"}, 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.

**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 semicolumns:

```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>)
```

`estimate_parameters(signal, x1, x2, only_current=False)`

Estimate the Donach by calculating the median (centre) and the variance parameter (sigma).

Note that an insufficient range will affect the accuracy of this method and that this method doesn’t estimate the asymmetry parameter (alpha).

**Parameters**

- **signal (Signal1D 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**  Returns True when the parameters estimation is successful

**Return type** bool
Examples

```python
>>> g = hs.model.components1D.Lorentzian()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:,:1] = 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)
```

hyperspy_components.eels_arctan module

```python
class hyperspy_components.eels_arctan.ArcTan(minimum_at_zero=False, **kwargs)
    Bases: hyperspy_components.expression.Expression

This is the legacy Arctan component dedicated to EELS measurements that will renamed to EELSArctan in v2.0.

To use the new Arctan component set minimum_at_zero=False. See the documentation of hyperspy_components.arctan.ArcTan() for details on the usage.

The EELS version hyperspy_components.eels_arctan.EELSArctan() (minimum_at_zero=True) shifts the function by A in the y direction.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression**(str) – Component function in SymPy text expression format with substitutions separated by ;. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- **name**(str) – Name of the component.

- **position**(str, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).

- **module**({"numpy", "numexpr", "scipy"}, 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.

---

**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 semicolumns:

```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>)
```

**class hyperspy._components.eels_arctan.EELSArctan(A=1.0, k=1.0, x0=1.0, module=['numpy', 'scipy'], **kwargs)**

Bases: **hyperspy._components.expression.Expression**

Arctan function component for EELS (with minimum at zero).

\[
f(x) = A \cdot \left(\frac{\pi}{2} + \arctan |k(x - x_0)|\right)
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>x0</td>
<td>x0</td>
</tr>
</tbody>
</table>

**Parameters**

• **A** *(float)* – Amplitude parameter. \(\lim_{x \to -\infty} f(x) = 0\) and \(\lim_{x \to \infty} f(x) = 2A\)

• **k** *(float)* – Slope (steepness of the step). The larger \(k\), the sharper the step.

• **x0** *(float)* – Center parameter \((f(x_0) = A)\).
Create a component from a string expression.
It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression (str)** – Component function in SymPy text expression format with substitutions separated by `. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables `x` and `y` must be included for 1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- **name (str)** – Name of the component.

- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).

- **module ({"numpy", "numexpr", "scipy"}, 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.

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>)
```

**hyperspy._components.eels_cl_edge module**

```python
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**

```
_calculate_effective_angle()
_calculate_knots()
_fine_structure_smoothing = 0.3
_get_E0()
_get_collection_angle()
_get_convergence_angle()
_get_fine_structure_active()
_get_fine_structure_width()
_integrate_GOS()
_onset_energy()
_set_E0(arg)
_set_collection_angle(arg)
_set_convergence_angle(arg)
_set_fine_structure_active(arg)
_set_fine_structure_coeff()
_set_fine_structure_width(arg)
```

**property collection_angle**

**property convergence_angle**

**property fine_structure_active**

**property 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

\[ \text{grad\_intensity}(E) \]

\[ \text{gui}(\text{display}=\text{True}, \text{toolkit}=\text{None}, **\text{kwargs}) \]

Display or return interactive GUI element if available.

Parameters

- \textbf{display} (bool) – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

- \textbf{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.

\[ \text{set\_microscope\_parameters}(E_0, \alpha, \beta, \text{energy\_scale}) \]

Parameters

- \textbf{E0} (float) – Electron beam energy in keV.

- \textbf{alpha} (float) – Convergence semi-angle in mrad.

- \textbf{beta} (float) – Collection semi-angle in mrad.

- \textbf{energy\_scale} (float) – The energy step in eV.

\[ \text{txt\_to\_fine\_structure\_coeff}(\text{filename}) \]

\textbf{hyperspy\_components\_eels\_double\_power\_law module}

\textbf{class} hyperspy\_components\_eels\_double\_power\_law\_DoublePowerLaw (\text{A=1e-05}, \text{r=3.0}, \text{origin=0.0}, \text{shift=20.0}, \text{ratio=1.0}, \text{left\_cutoff=0.0}, \text{module=’numexpr’}, \text{compute\_gradients=False}, **\text{kwargs})

\textbf{Bases: hyperspy\_components\_expression\_Expression}

Double power law component for EELS spectra.

\[ f(x) = A \cdot \left[ s_r \cdot (x - x_0 - x_s)^{-r} + (x - x_0)^{-r} \right] \]

<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.

**left_cutoff parameter can be used to set a lower threshold from which (The)**

**component will return 0. (The)**

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression (str)** – Component function in SymPy text expression format with substitutions separated by `. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name (str)** – Name of the component.
- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).
- **module ({"numpy", "numexpr", "scipy"}, 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.
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>)
```

**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.

**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

```python
class hyperspy._components.error_function.Erf(A=1.0, sigma=1.0, origin=0.0, module=['numpy', 'scipy'], **kwargs):
    Bases: hyperspy._components.expression.Expression

    Error function component.

    $f(x) = \frac{A}{2} \text{erf} \left( \frac{x - x_0}{\sqrt{2}\sigma} \right)$

    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.
```

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters
----------
- **expression** (str) – Component function in SymPy text expression format with substitutions separated by . See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** (str) – Name of the component.
- **position** (str, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).
- **module** ("numpy", "numexpr", "scipy") or “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.

---

**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>)
```

### hyperspy._components.exponential module

**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.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

**Parameters**

- `expression (str)` – Component function in SymPy text expression format with substitutions separated by `. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables `x` and `y` must be included for 1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- `name (str)` – Name of the component.

- `position (str, optional)` – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).

- `module ("numpy", "numexpr", "scipy", 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.

**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>)
```

`estimate_parameters(signal, x1, x2, only_current=False)`

Estimate the parameters for the exponential component by splitting the signal window into two regions and using their geometric means

Parameters

- **signal** (*BaseSignal 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

`hyperspy_components.expression module`

```python
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.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (*str*) – Component function in SymPy text expression format with substitutions separated by ;. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for
1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- **name (str)** – Name of the component.
- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).
- **module ("numpy", "numexpr", "scipy"), 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.

**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
```

(continues on next page)
(continued from previous page)

```python
(<Parameter one of my function component>,
 <Parameter two of my function component>)
```

**compile_function** *(module='numpy', position=False)*

Compile the function and calculate the gradient automatically when possible. Useful to recompile the function and gradient with a different module.

**function Nd** *(args)*

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly obtain the fitted component without iterating over the navigation axes.

```python
hyperspy._components.expression._fill_function_args(fn)
hyperspy._components.expression._fill_function_args_2d(fn)
```

**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>(\text{sigma})</td>
</tr>
<tr>
<td>(x_0)</td>
<td>(\text{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 \(\text{fwhm}\) and \(\text{height}\) attributes can be used to get and set the full-with-half-maximum and height of the distribution, respectively.

**See also:**

hyperspy._components.gaussianhf.GaussianHF

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

**Parameters**
• **expression** (*str*) – Component function in SymPy text expression format with substitutions separated by ;. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

• **name** (*str*) – Name of the component.

• **position** (*str*, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).

• **module** ({"numpy", "numexpr", "scipy"}, 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.

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 semicolumns:

```python
>>> expr = 'A*B/(A+B) ; A = sin(x)+one; B = cos(y) - two; y = tan(x)'
```

```python
>>> comp = hs.model.components1D.Expression(continues on next page)
```
expression=expr,
name='my function')

>>> comp.parameters
(<Parameter one of my function component>,
<Parameter two of my function component>)

**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[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
```

**property fwhm**

**property height**

hyperspy._components.gaussian._estimate_gaussian_parameters *(signal, x1, x2, only_current)*

**hyperspy._components.gaussian2d module**

**class hyperspy._components.gaussian2d.Gaussian2D** *(A=1.0, sigma_x=1.0, sigma_y=1.0, centre_x=0.0, centre_y=0.0, module='numexpr', **kwargs)*

**Bases:** hyperspy._components.expression.Expression

Normalized 2D elliptical Gaussian function component.

\[ f(x, y) = \frac{A}{2\pi s_x s_y} \exp \left[ -\frac{(x - x_0)^2}{2s_x^2} - \frac{(y - y_0)^2}{2s_y^2} \right] \]
### Parameters

- **$A$** *(float)* – Amplitude (height of the peak scaled by $2\pi s_x s_y$).
- **$s_x, s_y$** *(float)* – Width (scale parameter) of the Gaussian distribution in $x$ direction.
- **$sigma_x, sigma_y$** *(float)* – Width (scale parameter) of the Gaussian distribution in $y$ direction.
- **$centre_x, centre_y$** *(float)* – Location of the Gaussian maximum (peak position) in $x$ and $y$ directions.
- **add_rotation** *(bool)* – If True, add the parameter $rotation_angle$ corresponding to the angle between the $x$ and the horizontal axis.

#### fwhm_x, fwhm_x

Convenience attributes to get and set the full-with-half-maxima along the two axes.

**Type** float

### Properties

**sigma_major**
The sigma value of the major axis (axis with the largest sigma value).

**Type** float

**sigma_minor**
The sigma value of the minor axis (axis with the smallest sigma value).

**Type** float

**ellipticity**
Ratio between the major and minor axis.

**Type** float

**rotation_major_axis**
Rotation angle in radian between the major axis (axis with the largest sigma value) and the horizontal axis. Only for Gaussian2D component created with $add_rotation=True$.

**Type** float

**rotation_angle_wrapped**
Rotation angle in radian wrapped to $[0, 2\pi]$. Only for Gaussian2D component created with $add_rotation=True$.

**Type** float

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

**Parameters**

- **expression** *(str)* – Component function in SymPy text expression format with substitutions separated by `;`. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables $x$ and $y$ must be included for
1D or 2D components. Also, if module is “numexpr” the functions are limited to those that
numexpr support. See its documentation for details.

- **name (str)** – Name of the component.
- **position (str, optional)** – The parameter name that defines the position of the
  component if applicable. It enables interactive adjustment of the position of the component in
  the model. For 2D components, a tuple must be passed with the name of the two parameters
e.g. (“x0”, “y0”).
- **module ("numpy", "numexpr", "scipy"), 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 co-
  incide 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.

**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)
```

```python
>>> comp = hs.model.components1D.Expression(
    ...
    expression=expr,
    ...
    name='my function')
>>> comp.parameters
```

(continues on next page)
property ellipticity
property fwhm_x
property fwhm_y

property rotation_angle_wrapped
Angle in radian wrapped from 0 to 2*pi.

property rotation_major_axis
Angle between major axis and x-axis, in radian.

property sigma_major
property sigma_minor

hyperspy._components.gaussianhf module

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

Create a component from a string expression.
HyperSpy Documentation, Release 1.6.0

It automatically generates the partial derivatives and the class docstring.
Parameters
• expression (str) – Component function in SymPy text expression format with substitutions separated by ;. See examples and the SymPy documentation for details. In order to
vary the components along the signal dimensions, the variables x and y must be included for
1D or 2D components. Also, if module is “numexpr” the functions are limited to those that
numexpr support. See its documentation for details.
• name (str) – Name of the component.
• position (str, optional) – The parameter name that defines the position of the
component if applicable. It enables interative 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", "scipy"}, 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.
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:
>>>
...
...
...
...
...
...

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 semicolumns:

262

Chapter 3. hyperspy


>>> 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>)

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**

**class hyperspy_components.heaviside.HeavisideStep** *(A=1.0, n=0.0, module='numpy',
compute_gradients=False,
**kwargs)*

**Bases:** hyperspy_components.expression.Expression
The Heaviside step function.

\[ f(x) = \begin{cases} 
0 & x < n \\
A & x \geq n 
\end{cases} \]

Parameters

- **n** *(float)* – Location parameter defining the x position of the step.
- **A** *(float)* – Height parameter for x \( \geq n \).
- **kwargs** – Extra keyword arguments are passed to the Expression component.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** *(str)* – Component function in SymPy text expression format with substitutions separated by `. `. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \( x \) and \( y \) must be included for 1D or 2D components. Also, if **module** is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** *(str)* – Name of the component.
- **position** *(str, optional)* – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. ("x0", "y0").
- **module** *("numpy", "numexpr", "scipy"), 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.

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>)
```

\[ \frac{A B}{A + B} \]

\[ f(x) = \frac{a}{1 + b \cdot \exp \left[ -c \left( (x - x_0) \right) \right]} \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>x_0</td>
<td>origin</td>
</tr>
</tbody>
</table>

**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
- **origin** *(Float)* – Position of the sigmoid’s midpoint

3.1. hyperspy package 265
• **kwargs – Extra keyword arguments are passed to the Expression component.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

**Parameters**

- **expression (str)** – Component function in SymPy text expression format with substitutions separated by `. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables x and y must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- **name (str)** – Name of the component.

- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables iterative 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", "scipy", 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.

**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>)
```

hyperspy_components.lorentzian module

class hyperspy_components.lorentzian.Lorentzian

Bases: hyperspy_components.expression.Expression

Cauchy-Lorentz distribution (a.k.a. Lorentzian function) component.

\[
f(x) = \frac{A}{\pi} \left( \frac{\gamma}{(x - x_0)^2 + \gamma^2} \right)
\]

<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.
- **gamma (float)** – Scale parameter corresponding to the half-width-at-half-maximum of the peak, which corresponds to the interquartile spread.
- **centre (float)** – Location of the peak maximum.
- ****kwargs – Extra keyword arguments are passed to the Expression component.

For convenience the fwhm and height attributes can be used to get and set the full-with-half-maximum and height of the distribution, respectively.

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”, “scipy”), 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.

---

**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="Comp",
...     ...     A=1,
...     ...     B=2,
...     ...     y=1,
... )
```
... expression=expr,
... name='my function')

```python
>>> comp.parameters
(<Parameter one of my function component>,
 <Parameter two of my function component>)
```

**estimate_parameters** *(signal, x1, x2, only_current=False)*

Estimate the Lorentzian by calculating the median (centre) and half the interquartile range (gamma).

Note that an insufficient range will affect the accuracy of this method.

**Parameters**

- **signal** (*Signal1D 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 gaussian.py and https://en.wikipedia.org/wiki/Cauchy_distribution

**Examples**

```python
>>> g = hs.model.components1D.Lorentzian()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:,:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager[-1].offset = -10
>>> s.axes_manager[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
```

**property fwhm**

**property height**

```python
hyperspy._components.lorentzian._estimate_lorentzian_parameters(signal, x1, x2, only_current)
```
hyperspy._components.offset module

class hyperspy._components.offset.Offset (offset=0.0)
    Bases: hyperspy.component.Component

    Component to add a constant value in the y-axis.

    \[ f(x) = k \]

    Parameters
    offset (float) –

    _function (x, o)

estimate_parameters (signal, x1, x2, only_current=False)
    Estimate the parameters by the two area method

    Parameters
    • signal (BaseSignal 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

    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.

    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)
    
    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._components.pes_voigt module

class hyperspy._components.pes_voigt.**PESVoigt**
    
    Voigt component for photoemission spectroscopy data analysis.

    Voigt profile component with support for shirley background, non_isochromaticity, transmission_function corrections and spin orbit splitting specially suited for photoemission spectroscopy data analysis.

    \[ f(x) = G(x) \cdot L(x) \]

    where \( G(x) \) is the Gaussian function and \( L(x) \) is the Lorentzian function. This component uses an approximate formula by David (see Notes).

    Parameters

    - **area** (Parameter) – Intensity below the peak.
    - **centre** (Parameter) – Location of the maximum of the peak.
    - **FWHM** (Parameter) – FWHM = \( 2\sigma \sqrt{(2 \log(2))} \) of the Gaussian distribution.
    - **gamma** (Parameter) – \( \gamma \) of the Lorentzian distribution.
    - **resolution** (Parameter) –
    - **shirley_background** (Parameter) –
    - **non_isochromaticity** (Parameter) –
transmission_function (Parameter) –
spin_orbit_splitting (Bool) –
spin_orbit_branching_ratio (float) –
spin_orbit_splitting_energy (float) –

Notes

doi:10.1107/S0021889886089999

estimate_parameters (signal, E1, E2, only_current=False)

Estimate the Voigt function by calculating the momenta of 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 Exit status required for the remove_background() function.

Return type bool

Notes

Adapted from http://www.scipy.org/Cookbook/FittingData

Examples

```python
>>> g = hs.model.components1D.PESVoigt()
>>> 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)
```

function (x)

This is the legacy Voigt profile component dedicated to photoemission spectroscopy data analysis that will renamed to PESVoigt in v2.0. To use the new Voigt lineshape component set legacy=False. See the documentation of hyperspy._components.voigt.Voigt() for details on the usage of the new Voigt component and hyperspy._components.pes_voigt.PESVoigt() for the legacy component.

\[ f(x) = G(x) \cdot L(x) \]

where \( G(x) \) is the Gaussian function and \( L(x) \) is the Lorentzian function. This component uses an approximate formula by David (see Notes).
Notes


property FWHM
property gwidth
property lwidth

hyperspy._components.pes_voigt.voigt(x, FWHM=1, gamma=1, center=0, scale=1)

Voigt lineshape.

The voigt peak is the convolution of a Lorentz peak with a Gaussian peak:

\[ f(x) = G(x) \cdot L(x) \]

where \( G(x) \) is the Gaussian function and \( L(x) \) is the Lorentzian function. In this case using an approximate formula by David (see Notes). This approximation improves on the pseudo-Voigt function (linear combination instead of convolution of the distributions) and is, to a very good approximation, equivalent to a Voigt function:

\[ z(x) = \frac{x + i\gamma}{\sqrt{2}\sigma} \]
\[ w(z) = \frac{e^{-z^2} \text{erfc}(-iz)}{\sqrt{2\pi}\sigma} \]
\[ f(x) = A \cdot \Re\{w[z(x - x_0)]\} \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_0 )</td>
<td>center</td>
</tr>
<tr>
<td>( A )</td>
<td>scale</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>gamma</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>sigma</td>
</tr>
</tbody>
</table>

Parameters

- **gamma** *(real)* – The half-width half-maximum of the Lorentzian.
- **FWHM** *(real)* – The FWHM = \( 2\sigma \sqrt{(2\log(2))} \) of the Gaussian.
- **center** *(real)* – Location of the center of the peak.
- **scale** *(real)* – Value at the highest point of the peak.

Notes

hyperspy._components.polynomial module

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.

\[ f(x) = a_2 x^2 + a_1 x^1 + a_0 \]

Zero padding is used for polynomial of order > 10.

Parameters

- **order** (int) – Order of the polynomial, must be different from 0.
- ****kwargs – Keyword arguments can be used to initialise the value of the parameters, i.e. a2=2, a1=3, a0=1.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (str) – Component function in SymPy text expression format with substitutions separated by \( ; \). See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \( x \) and \( y \) must be included for 1D or 2D components. Also, if **module** is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** (str) – Name of the component.
- **position** (str, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. (“x0”, “y0”).
- **module** ("numpy", "numexpr", "scipy"), 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.
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 semicolumns:

```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
(\text{Parameter one of my function component},
\text{Parameter two of my function component})
```

`estimate_parameters(signal, x1, x2, only_current=False)`

Estimate the parameters by the two area method

- **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
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 length the order of the polynomial. For example, the [1,2,3] coefficients define the following 3rd order polynomial: \( f(x) = 1x^2 + 2x + 3 \)

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.

_function (x, coefficients)

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

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

```python
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.

\[ f(x) = A \cdot (x - x_0)^{-r} \]

<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>
</tbody>
</table>

Parameters

- **A (float)** – Height parameter.
- **r (float)** – Power law coefficient.
- **origin (float)** – Location parameter.
- ****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.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression (str)** – Component function in SymPy text expression format with substitutions separated by ;. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \(x\) and \(y\) must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name (str)** – Name of the component.
- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables iterative 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", "scipy"), 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.

**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>)
```

**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 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)*

  **grad_A** *(x)*
HyperSpy Documentation, Release 1.6.0

grad_origin(x)
grad_r(x)

hyperspy_components.rc module

class hyperspy_components.rc.RC(Vmax=1.0, V0=0.0, tau=1.0, module='numexpr', **kwargs)
Bases: 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_max</td>
<td>Vmax</td>
</tr>
<tr>
<td>V_0</td>
<td>V0</td>
</tr>
<tr>
<td>\tau</td>
<td>tau</td>
</tr>
</tbody>
</table>

Parameters

- **Vmax (float)** – maximum voltage, asymptote of the function for \(\lim_{x \to \infty}\)
- **V0 (float)** – vertical offset
- **tau (float)** – \(\tau=RC\) is the RC circuit time constant (voltage rise time)
- ****kwargs – Extra keyword arguments are passed to the Expression component.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression (str)** – Component function in SymPy text expression format with substitutions separated by \\(.\). See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \(x\) and \(y\) must be included for 1D or 2D components. Also, if module is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name (str)** – Name of the component.
- **position (str, optional)** – The parameter name that defines the position of the component if applicable. It enables iterative 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", "scipy"), 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.

3.1. hyperspy package
• **rename pars** *(dictionary)* – The desired name of a parameter may sometimes co-
incide 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 automati-
cally 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.

**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 semicolumns:

```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>)
```

**hyperspy._components.scalable_fixed_pattern module**

```python
class hyperspy._components.scalable_fixed_pattern.ScalableFixedPattern(signal1D,
yscale=1.0,
xscale=1.0,
shift=0.0,
interpolate=True)
```

**Bases:** *hyperspy.component.Component*

Fixed pattern component with interpolation support.

\[ f(x) = a \cdot s (b \cdot x - x_0) + c \]
The fixed pattern is defined by a single spectrum which must be provided to the ScalableFixedPattern constructor, e.g.:

```
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=['numpy', '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/

\[
f(x) = 2A\phi(x)\Phi(x)
\]

\[
\phi(x) = \frac{1}{\sqrt{2\pi}}\exp\left[-\frac{(x-x_0)^2}{2}\right]
\]

\[
\Phi(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{\alpha t(x)}{\sqrt{2}}\right)\right]
\]

\[
t(x) = \frac{x-x_0}{\omega}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</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

- **x0 (float)** – Location of the peak position (not maximum, which is given by the mode property).
- **A (float)** – Height parameter of the peak.
- **scale (float)** – Width (sigma) parameter.
- **shape (float)** – Skewness (asymmetry) parameter. For shape=0, the normal distribution (Gaussian) is obtained. The distribution is right skewed (longer tail to the right) if shape>0 and is left skewed if shape<0.

The properties mean (position), variance, skewness and mode (=position of maximum) are defined for convenience.

Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters
• **expression** (**str**) – Component function in SymPy text expression format with substitutions separated by `:`. See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables `x` and `y` must be included for 1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

• **name** (**str**) – Name of the component.

• **position** (**str, optional**) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters `e.g. ("x0", "y0")`.

• **module** (**"numpy", "numexpr", "scipy"**, 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.

**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)'
```

```python
>>> comp = hs.model.components1D.Expression(
```

(continues on next page)
estimate_parameters (signal, x1, x2, only_current=False)

Estimate the skew normal distribution 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


Examples

```python
>>> g = hs.model.components1D.SkewNormal()
>>> x = np.arange(-10, 10, 0.01)
>>> data = np.zeros((32, 32, 2000))
>>> data[:] = g.function(x).reshape((1, 1, 2000))
>>> s = hs.signals.Signal1D(data)
>>> s.axes_manager._axes[-1].offset = -10
>>> s.axes_manager._axes[-1].scale = 0.01
>>> g.estimate_parameters(s, -10, 10, False)
```

property mean

property mode

property skewness

property variance

hyperspy._components.skew_normal._estimate_skewnormal_parameters (signal, x1, x2, only_current)
**hyperspy._components.split_pvoigt module**

```python
class hyperspy._components.split_pvoigt.SplitVoigt (A=1.0, sigma1=1.0, sigma2=1.0, fraction=0.0, centre=0.0)

Bases: hyperspy.component.Component

Split pseudo-Voigt

\[ pV(x, centre, \sigma) = (1 - \eta)G(x, centre, \sigma) + \eta L(x, centre, \sigma) \]

\[ f(x) = \begin{cases} 
    pV(x, centre, \sigma_1), & x \leq centre \\
    pV(x, centre, \sigma_2), & x > centre 
\end{cases} \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>\eta</td>
<td>fraction</td>
</tr>
<tr>
<td>\sigma_1</td>
<td>sigma1</td>
</tr>
<tr>
<td>\sigma_2</td>
<td>sigma2</td>
</tr>
<tr>
<td>centre</td>
<td>centre</td>
</tr>
</tbody>
</table>
```

**Note:** This is a Voigt function in which the upstream and downstream variance or sigma is allowed to vary to create an asymmetric profile. In this case, the Voigt is a pseudo-Voigt consisting of a mixed Gaussian and Lorentzian sum.

```python
estimate_parameters(signal, x1, x2, only_current=False)
```

Estimate the split 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](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)
```

**function** *(x)*

Split pseudo voigt - a linear combination of gaussian and lorentzian

**Parameters**

- **x** *(array)* – independent variable
- **A** *(float)* – area of pvoigt peak
- **center** *(float)* – center position
- **sigma1** *(float)* – standard deviation <= center position
- **sigma2** *(float)* – standard deviation > center position
- **fraction** *(float)* – weight for lorentzian peak in the linear combination, and (1-fraction) is the weight for gaussian peak.

**function_nd** *(axis)*

Returns a numpy array containing the value of the component for all indices. If enough memory is available, this is useful to quickly obtain the fitted component without iterating over the navigation axes.

**property height**

**hyperspy._components.voigt module**

**class hyperspy._components.voigt.Voigt** *(centre=10.0, area=1.0, gamma=0.2, sigma=0.1, module=['numpy', 'scipy'], **kwargs)*

**Bases:** hyperspy._components.expression.Expression

Voigt component.

Symmetric peak shape based on the convolution of a Lorentzian and Normal (Gaussian) distribution:

\[ f(x) = G(x) \cdot L(x) \]

where \( G(x) \) is the Gaussian function and \( L(x) \) is the Lorentzian function. In this case using an approximate formula by David (see Notes). This approximation improves on the pseudo-Voigt function (linear combination instead of convolution of the distributions) and is, to a very good approximation, equivalent to a Voigt function:

\[
z(x) = \frac{x + i\gamma}{\sqrt{2\sigma}}
\]
\[
w(z) = \frac{e^{-z^2} \text{erfc}(-iz)}{\sqrt{2\pi\sigma}}
\]
\[
f(x) = A \cdot \Re \{w[z(x - x_0)]\} \]

286 Chapter 3. hyperspy
HyperSpy Documentation, Release 1.6.0

<table>
<thead>
<tr>
<th>Variable</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>centre</td>
</tr>
<tr>
<td>$A$</td>
<td>area</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>gamma</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>sigma</td>
</tr>
</tbody>
</table>

Parameters

- **centre** *(float)* – Location of the maximum of the peak.
- **area** *(float)* – Intensity below the peak.
- **gamma** *(float)* – $\gamma = \text{HWHM of the Lorentzian distribution}$.
- **sigma** *(float)* – $2\sigma \sqrt{2 \log(2)} = \text{FWHM of the Gaussian distribution}$.

For convenience the `gwidth` and `lwidth` attributes can also be used to set and get the FWHM of the Gaussian and Lorentzian parts of the distribution, respectively. For backwards compatibility, `FWHM` is another alias for the Gaussian width.

Notes


Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** *(str)* – Component function in SymPy text expression format with substitutions separated by `. `See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables $x$ and $y$ must be included for 1D or 2D components. Also, if `module` is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.
- **name** *(str)* – Name of the component.
- **position** *(str, optional)* – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. ("x0", "y0").
- **module** *("numpy", "numexpr", "scipy"), 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"}.

3.1. hyperspy package 287
• **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 – Keyphrase arguments can be used to initialise the value of the parameters.

**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(-0.5*(x - x0)**2 * fwhm**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>)
```

**property FWHM**

**estimate_parameters**(signal, x1, x2, only_current=False)

Estimate the Voigt function by calculating the momenta of 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**  Exit status required for the remove_background() function.

**Return type**  bool
Notes

Adapted from http://www.scipy.org/Cookbook/FittingData

Examples

```python
>>> g = hs.model.components1D.Voigt(legacy=False)
>>> 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)
```

```plaintext
property gwidth
property lwidth
```

**hyperspy_components.volume_plasmon_drude module**

class hyperspy_components.volume_plasmon_drude.VolumePlasmonDrude (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


Create a component from a string expression.

It automatically generates the partial derivatives and the class docstring.

Parameters

- **expression** (str) – Component function in SymPy text expression format with substitutions separated by \( : \). See examples and the SymPy documentation for details. In order to vary the components along the signal dimensions, the variables \( x \) and \( y \) must be included for 1D or 2D components. Also, if **module** is “numexpr” the functions are limited to those that numexpr support. See its documentation for details.

- **name** (str) – Name of the component.

- **position** (str, optional) – The parameter name that defines the position of the component if applicable. It enables interactive adjustment of the position of the component in the model. For 2D components, a tuple must be passed with the name of the two parameters e.g. \( (“x0”, “y0”) \).

- **module**(\{"numpy", "numexpr", "scipy"\}, 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.

**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>)
```

$$\text{grad_fwhm}(x)$$

$$\text{grad_intensity}(x)$$

$$\text{grad_plasmon_energy}(x)$$

Module contents

hypserspy_signals package

Submodules

hypserspy_signals.common_signal1d module

```python
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
```

3.1. hyperspy package
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)

   _get_amplitude()
   _get_phase()
   _set_amplitude(amplitude)
   _set_imag(imag)
   _set_phase(phase)
   _set_real(real)

   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

   argand_diagram(size=[256, 256], range=None)
      Calculate and plot Argand diagram of complex signal

      Parameters

      • size ([int, int], optional) – Size of the Argand plot in pixels (Default: [256, 256])
• `range (array_like, shape(2,2) or shape(2,) optional)` – The position of the edges of the diagram (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram. (Default: None)

**Returns**  Argand diagram as Signal2D

**Return type**  argand_diagram

**Examples**

```python
>>> import hyperspy.api as hs
>>> holo = hs.datasets.example_signals.object_hologram()
>>> ref = hs.datasets.example_signals.reference_hologram()
>>> w = holo.reconstruct_phase(ref)
>>> w.argand_diagram(range=[-3, 3]).plot()
```

```python
class hyperspy._signals.complex_signal.ComplexSignal_mixin(*args, **kwargs)
Bases: object
BaseSignal subclass for complex data.

_dtype = 'complex'

_get_amplitude(amplitude)

_get_imag()

_get_phase(phase)

_get_real()

property amplitude
  Get/set the amplitude of the data. Returns an appropriate HyperSpy signal.

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 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

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.

plot (power_spectrum=False, representation='cartesian', same_axes=True, fft_shift=False, navigators='auto', axes_manager=None, norm='auto', **kwargs)
  Plot the signal at the current coordinates.
```
For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

- **power_spectrum** *(bool, default 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.

- **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.

• **navigator_kwds** (dict) – Only for image navigator, additional keyword arguments for matplotlib.pyplot.imshow().

• ****kwargs** – Only when plotting an image: additional (optional) keyword arguments for matplotlib.pyplot.imshow().

**property real**

Get/set the real part of the data. Returns an appropriate HyperSpy signal.

**unwrapped_phase** (wrap_around=False, seed=None, show_progressbar=None, parallel=None, max_workers=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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

• **max_workers** (None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**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

_get_amplitude()

_get_phase()

_set_amplitude(amplitude)
HyperSpy Documentation, Release 1.6.0

_set_imag(imag)
_set_phase(phase)
_set_real(real)

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

argand_diagram(*args, **kwargs)
  Calculate and plot Argand diagram of complex signal

  Parameters
  size ([int, int], optional) – Size of the Argand plot in pixels (Default: [256, 256])

  range (array_like, shape(2, 2) or shape(2,) optional) – The position of the edges of the diagram (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram. (Default: None)

  Returns
  Argand diagram as Signal2D

  Return type
  argand_diagram

Examples

```python
>>> import hyperspy.api as hs
>>> holo = hs.datasets.example_signals.object_hologram()
>>> ref = hs.datasets.example_signals.reference_hologram()
>>> w = holo.reconstruct_phase(ref)
>>> w.argand_diagram(range=[-3, 3]).plot()
```

hyperspy._signals.complex_signal.format_title(thing)

hyperspy._signals.complex_signal1d module

class hyperspy._signals.complex_signal1d.ComplexSignal1D(*args, **kwargs)
  BaseSignal subclass for complex 1-dimensional data.

  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.

```python
_signal_dimension = 1
```

**class** hyperspy._signals.complex_signal1d.LazyComplexSignal1D(*args, **kwargs)

**Bases:** hyperspy._signals.complex_signal1d.ComplexSignal1D, hyperspy._signals.complex_signal.LazyComplexSignal

BaseSignal subclass for lazy complex 1-dimensional data.

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.

```python
_signal_dimension = 1
```

**hyperspy_signals.complex_signal2d module**

**class** hyperspy._signals.complex_signal2d.Complex2Dmixin(*args, **kw)

**Bases:** object

BaseSignal subclass for lazy complex 2-dimensional data.

```python
_signal_dimension = 2
```

**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.

```python
plot(power_spectrum=False, fft_shift=False, navigator='auto', plot_markers=True, autoscale='v', saturated_pixels=None, norm='auto', vmin=None, vmax=None, gamma=1.0, linthresh=0.01, linscale=0.1, scalebar=True, scalebar_color='white', axes_ticks=None, axes_off=False, axes_manager=None, no_nans=False, colorbar=True, centre_colormap='auto', min_aspect=0.1, **kwargs)
```

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimenion equal to 0, 1 and 2 can be plotted.

Parameters

- **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.
- **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.
- 'slider':
  - If `navigation dimension` > 0 a window with one slider per axis is raised to navigate the data.
- '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.

- `navigator_kwds` (dict) – Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

- `colorbar` (bool, optional) – If true, a colorbar is plotted for non-RGB images.

- `autoscale` (str) – The string must contain any combination of the ‘x’, ‘y’ and ‘v’ characters. If ‘x’ or ‘y’ are in the string, the corresponding axis limits are set to cover the full range of the data at a given position. If ‘v’ (for values) is in the string, the contrast of the image will be set automatically according to `vmin` and `vmax` when the data or navigation indices change. Default is ‘v’.

- `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.

- `deprecated`: (.) – 1.6.0: `saturated_pixels` will be removed in HyperSpy 2.0.0, it is replaced
  by `vmin`, `vmax` and `autoscale`.

- `norm` ("auto", "linear", "power", "log", "symlog" or a subclass of) – `matplotlib.colors.Normalise` Set the norm of the image to display. If “auto”, a linear scale is used except if when `power_spectrum=True` in case of complex data type. “symlog” can be used to display negative value on a negative scale - read `matplotlib.colors.SymLogNorm` and the `linthresh` and `linscale` parameter for more details.

- `vmax` (vmin,) – `vmin` and `vmax` are used to normalise the displayed data. It can be a float or a string. If string, it should be formatted as ‘xth’, where ‘x’ must be an float in the [0, 100] range. ‘x’ is used to compute the x-th percentile of the data. See `numpy.percentile()` for more information.

- `gamma` (float) – Parameter used in the power-law normalisation when the parameter `norm="power"`. Read `matplotlib.colors.PowerNorm` for more details. Default value is 1.0.

- `linthresh` (float) – When used with `norm="symlog"`, define the range within which the plot is linear (to avoid having the plot go to infinity around zero). Default value is 0.01.

- `linscale` (float) – This allows the linear range (-linthresh to linthresh) to be stretched relative to the logarithmic range. Its value is the number of powers of base to use for each
half of the linear range. See matplotlib.colors.SymLogNorm for more details. Default value is 0.1.

- **scalebar** *(bool, optional)* – If True and the units and scale of the x and y axes are the same a scale bar is plotted.

- **scalebar_color** *(str, optional)* – A valid MPL color string; will be used as the scalebar color.

- **axes_ticks** *(bool, optional)* – If True, plot the axes ticks. If None axes_ticks are only plotted when the scale bar is not plotted. If False the axes ticks are never plotted.

- **axes_off** *(bool)* – Default is False.

- **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 when plotting an image: additional (optional) keyword arguments for matplotlib.pyplot.imshow().

class hyperspy._signals.complex_signal2d.ComplexSignal2D(*args, **kw)

Bases: hyperspy._signals.complex_signal2d.Complex2Dmixin, hyperspy._signals.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_signal1d.LazyComplexSignal

BaseSignal subclass for lazy complex 2-dimensional data.

hyperspy._signals.dielectric_function module

class hyperspy._signals.dielectric_function.DielectricFunction(*args, **kwargs)

Bases: hyperspy._signals.dielectric_function.DielectricFunction_mixin, hyperspy._signals.complex_signal1d.ComplexSignal1D

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.

```python
class hyperspy._signals.dielectric_function.DielectricFunction_mixin
    Bases: object

    _alias_signal_types = ['dielectric function']
    _signal_type = 'DielectricFunction'

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*0), neff1 and neff2:

    \[ n_{\text{eff}1} = n_{\text{eff}} \left( -\Im \left( \epsilon - 1 \right) \right) \]

    and:

    \[ n_{\text{eff}2} = n_{\text{eff}} \left( \epsilon_2 \right) \]

    This method computes and return both.

    Parameters
    nat (float) – Number of atoms (or molecules) per unit volume of the sample.

    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
```

**Notes**

```python
class hyperspy._signals.dielectric_function.LazyDielectricFunction(*args, **kwargs)
    Bases: hyperspy._signals.dielectric_function.DielectricFunction, hyperspy._signals.complex_signal1d.LazyComplexSignal1D

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.
```

---

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_background_windows_markers(windows_position)
Plot the background windows associated with each X-ray lines.
For X-ray lines, a black line links the left and right window with the average value in each window.

Parameters windows_position (2D array of float) – 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.

See also: estimate_background_windows(), get_lines_intensity()

_add_vertical_lines_groups(position, **kwargs)
Add vertical markers for each group that shares the color.

Parameters

• position (2D array of float) – The position on the signal axis. Each row corresponds to a group.

• kwargs – keywords argument for markers.vertical_line

_get_beam_energy()
Get the beam energy.

The return value is in the same units than the signal axis

_get_line_energy(Xray_line, FWHM_MnKa=None)
Get the line energy and the energy resolution of a Xray line.

The return values are in the same units than the signal axis

Parameters

• Xray_line (strings) – Valid element X-ray lines e.g. Fe_Kb

• FWHM_MnKa ((None, float, 'auto')) – The energy resolution of the detector in eV if ‘auto’, used the one in ‘self.metadata.Acquisition_instrument.SEM.Detector.EDS.energy_resolution_MnKa’

Returns

• float (the line energy, if FWHM_MnKa is None)

• (float,float) (the line energy and the energy resolution, if FWHM_MnKa)

• is not None

_get_lines_from_elements(elements, only_one=False, only_lines='a')
Returns the X-ray lines of the given elements in spectral range of the data.

Parameters

• elements (list of strings) – A list containing the symbol of the chemical elements.
HyperSpy Documentation, Release 1.6.0

• **only_one** *(bool)* – If False, add all the lines of each element in the data spectral range. If True 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 returned.

Returns

Return type  list of X-ray lines alphabetically sorted

```
_get_xray_lines(xray_lines=None, only_one=None, only_lines='a')
```

```
_get_xray_lines_in_spectral_range(xray_lines)
```

Return the lines in the energy range

Parameters  **xray_lines** *(List of string)* – The xray_lines

Returns

Return type  The list of xray_lines in the energy range

```
_parse_xray_lines(xray_lines, only_one, only_lines)
```

```
_plot_xray_lines(xray_lines=False, only_lines='a', 'b', only_one=False, background_windows=None, integration_windows=None)
```

```
_signal_type = 'EDS'
```

```
_xray_marker_closed(obj)
```

```
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

```
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> print(s.metadata.Sample.elements)
['Al', 'Ar', 'C', 'Cu', 'Mn', 'Zr']

>>> s.add_elements(['Ar'])

>>> print(s.metadata.Sample.elements)
['Al', 'Ar', '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 use 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)
>>> 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

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> iw = s.estimate_integration_windows()
>>> s.plot(integration_windows=iw)
>>> s.get_lines_intensity(integration_windows=iw, plot_result=True)
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 Xray 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, Iterable* of strings)) – 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. * Note that while dictionaries and strings are iterable, their use is ambiguous and specifically not allowed.

- **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** *(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’ and ‘SEM.Stage.tilt_beta in ‘metadata’.

Returns take_off_angle – in Degree

Return type float

Examples

```python
>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.get_take_off_angle()
37.0
>>> s.set_microscope_parameters(tilt_stage=20.)
>>> s.get_take_off_angle()
57.0
```

See also:

hs.eds.take_off_angle()

plot(xray_lines=False, only_lines='a', 'b', only_one=False, background_windows=None, integration_windows=None, navigator='auto', plot_markers=True, autoscale='v', norm='auto', axes_manager=None, navigator_kwds={}, **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 an image navigator) and `navigation_shape` must be equal to the `navigator_shape` of the current object (for a dynamic navigator). If the signal `dtype` is RGB or RGBA this parameter has no effect and the value is always set to 'slider'.

• `axes_manager` (None or `AxesManager`) – If None, the signal’s `axes_manager` attribute is used.

• `plot_markers` (bool, default True) – Plot markers added using `s.add_marker(marker, permanent=True)`. Note, a large number of markers might lead to very slow plotting.

• `navigator_kwds` (dict) – Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

• `norm` (str, 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).

• `autoscale` (str) – The string must contain any combination of the 'x' and 'v' characters. If 'x' or 'v' (for values) are in the string, the corresponding horizontal or vertical axis
limits are set to their maxima and the axis limits will reset when the data or the navigation indices are changed. Default is ‘v’.

Examples

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

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot(True)

>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> s.add_lines()
>>> bw = s.estimate_background_windows()
>>> s.plot(background_windows=bw)

>>> s = hs.datasets.example_signals.EDS_SEM_Spectrum()
>>> s.plot(['Mn_Ka'], integration_windows='auto')

>>> 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

```python
>>> spectrum = hs.signals.EDSTEMSpectrum(np.ones([4, 4, 10]))
>>> spectrum.data[1, 2, 9] = 5
>>> print(spectrum)
<EDXTEMSpectrum, 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)
>>> s.set_lines(['Cu_Ka'])
>>> print(s.metadata.Sample.xray_lines)
['Al_Ka', 'C_Ka', 'Cu_La', 'Mn_La', 'Zr_La']
['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)
```

**class** `hyperspy._signals.eds.LazyEDSSpectrum(*args, **kwargs)`

**Bases:** `hyperspy._signals.eds.EDSSpectrum`, `hyperspy._signals.signal1d.LazySignal1D`
class hyperspy._signals.eds_sem.EDSSEMParametersUI(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.SEM.Detector.EDS.azimuth_angle': 'azimuth_angle', ...

class hyperspy._signals.eds_sem.EDSSEMSpectrum(*args, **kwargs)
    Bases: hyperspy._signals.eds_sem.EDSSEM_mixin, hyperspy._signals.eds.EDSSpectrum

class hyperspy._signals.eds_sem.EDSSEM_mixin(*args, **kwargs)
    Bases: object

    _are_microscope_parameters_missing()
    Check if the EDS parameters necessary for quantification are defined in metadata. If not, in interactive mode raises an UI item to fill the values

    _load_from_TEM_param()
    Transfer metadata.Acquisition_instrument.TEM to metadata.Acquisition_instrument.SEM

    _set_default_param()
    Set to value to default (defined in preferences)

    _signal_type = 'EDS_SEM'

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 EDSSEMMModel instance.

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.EDSSSEMSpectrum(
    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', ... 'Acquisition_instrument.TEM.beam_energy': 'beam_energy', 'Acquisition_instrument.TEM.probe_area': 'probe_area' }

class hyperspy._signals.eds_tem.EDSTEMSpectrum(*args, **kwargs)
Bases: hyperspy._signals.eds_tem.EDSTEM_mixin, hyperspy._signals.eds.EDSSpectrum

class hyperspy._signals.eds_tem.EDSTEM_mixin(*args, **kwargs)
Bases: object

CL_get_mass_thickness (weight_percent, thickness)
   Creates a array of mass_thickness based on a known material composition and measured thickness. Required for absorption correction calculations using the Cliff Lorimer method.

   Input given by i*t*N, i the current, t the acquisition time, and N the number of electron by unit electric charge.

   Parameters
   • composition (stack of compositions as determined from an initial k_factor) – quantification. Probe current in nA
   • thickness (float or array) – Either a float value for thickness in nm or an array equal to the size of the EDX map with thickness at each position of the sample.

   Returns
   Return type Mass_thickness as an array in kg/m².

_are_microscope_parameters_missing ()
   Check if the EDS parameters necessary for quantification are defined in metadata.

_get_dose (method, beam_current='auto', live_time='auto', probe_area='auto')
   Calculates the total electron dose for the zeta-factor or cross section methods of quantification.

   Input given by i*t*N, i the current, t the acquisition time, and N the number of electron by unit electric charge.

   Parameters
• **method** ('zeta' or 'cross_section') – If ‘zeta’, the dose is given by \(i t N\) If ‘cross section’, the dose is given by \(i t N/A\) where \(i\) is the beam current, \(t\) is the acquisition time, \(N\) is the number of electrons per unit charge (1/e) and \(A\) is the illuminated beam area or pixel area.

• **beam_current** (float) – Probe current in nA

• **live_time** (float) – Acquisition time in s, compensated for the dead time of the detector.

• **probe_area** (float or 'auto') – The illumination area of the electron beam in \(\text{nm}^2\). If ‘auto’ the value is extracted from the scale axes_manager. Therefore we assume the probe is oversampling such that the illumination area can be approximated to the pixel area of the spectrum image.

**Returns**

**Return type** Dose in electrons (zeta factor) or electrons per \(\text{nm}^2\) (cross_section)

See also:

`set_microscope_parameters()`

`_set_default_param()`

Set to value to default (defined in preferences)

`_signal_type` = 'EDS_TEM'

`create_model` (auto_background=True, auto_add_lines=True, *args, **kwargs)

Create a model for the current TEM EDS data.

**Parameters**

• **auto_background** (bool, default True) – If True, adds automatically a polynomial order 6 to the model, using the edsmode.add_polynomial_background method.

• **auto_add_lines** (bool, default True) – If True, automatically add Gaussians for all X-rays generated in the energy range by an element using the edsmode.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)

Apply a decomposition to a dataset with a choice of algorithms.

The results are stored in `self.learning_results`.

Read more in the `User Guide`.

**Parameters**

• **normalize_poissonian_noise** (bool, default True) – If True, scale the signal to normalize Poissonian noise using the approach described in [Keenan2004].

• **navigation_mask** (None or float or boolean numpy array, default 1.0) – 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, default True)* – If true, applied a morphologic closing to the mask obtained by vacuum_mask.

• **algorithm** *("SVD", "MLPCA", "sklearn_pca", "NMF", "sparse_pca", "mini_batch_sparse_pca", "RPCA", "ORPCA", "ORNMF", custom object), default "SVD")* – The decomposition algorithm to use. If algorithm is an object, it must implement a fit_transform() method or fit() and transform() methods, in the same manner as a scikit-learn estimator.

• **output_dimension** *(None or int)* – Number of components to keep/calculate. Default is None, i.e. \min(data.shape).

• **centre** *(None, "navigation", "signal"), default None)* –
  - If None, the data is not centered prior to decomposition.
  - If “navigation”, the data is centered along the navigation axis. Only used by the “SVD” algorithm.
  - If “signal”, the data is centered along the signal axis. Only used by the “SVD” algorithm.

• **auto_transpose** *(bool, default True)* – If True, automatically transposes the data to boost performance. Only used by the “SVD” algorithm.

• **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. Only used by the “MLPCA” algorithm.

• **var_func** *(None or function or numpy array, default None)* –
  - If None, ignored
  - If function, applies the function to the data to obtain var_array. Only used by the “MLPCA” algorithm.
  - If numpy array, creates var_array by applying a polynomial function defined by the array of coefficients to the data. Only used by the “MLPCA” algorithm.

• **reproject** *(None, "signal", "navigation", "both"), default None)* – If not None, the results of the decomposition will be projected in the selected masked area.

• **return_info** *(bool, default False)* – The result of the decomposition is stored internally. However, some algorithms generate some extra information that is not stored. If True, return any extra information if available. In the case of sklearn.decomposition objects, this includes the sklearn Estimator object.

• **print_info** *(bool, default True)* – If True, print information about the decomposition being performed. In the case of sklearn.decomposition objects, this includes the values of all arguments of the chosen sklearn algorithm.

• **svd_solver** *("auto", "full", "arpack", "randomized"), default "auto")* –
  If auto: The solver is selected by a default policy based on data.shape and output_dimension: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.
If full: run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing.

If arpack: use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly $0 < \text{output\_dimension} < \text{min(data.shape)}$.

If randomized: use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components.

- **copy** *(bool, default True)* –
  - If True, stores a copy of the data before any pre-treatments such as normalization in `s._data_before_treatments`. The original data can then be restored by calling `s.undo_treatments()`.
  - If False, no copy is made. This can be beneficial for memory usage, but care must be taken since data will be overwritten.

- **kwargs** *(extra keyword arguments)* – Any keyword arguments are passed to the decomposition algorithm.

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
```

`get_probe_area` *(navigation_axes=None)*
Calculates a pixel area which can be approximated to probe area, when the beam is larger than or equal to pixel size. The probe area can be calculated only when the number of navigation dimension are less than 2 and all the units have the dimensions of length.
Parameters

**navigation_axes** *(DataAxis, string or integer (or list of))* – Navigation axes corresponding to the probe area. If string or integer, the provided value is used to index the axes_manager.

Returns

**Return type** probe area in nm².

Examples

```python
>>> s = hs.datasets.example_signals.EDS_TEM_Spectrum()
>>> si = hs.stack([s] * 3)
>>> si.axes_manager.navigation_axes[0].scale = 0.01
>>> si.axes_manager.navigation_axes[0].units = 'm'
>>> si.get_probe_area()
100.0
```

**quantification** *(intensities, method, factors, composition_units='atomic', absorption_correction=False, take_off_angle='auto', thickness='auto', convergence_criterion=0.5, navigation_mask=1.0, closing=True, plot_result=False, probe_area='auto', max_iterations=30, **kwargs)*

Absorption corrected quantification using Cliff-Lorimer, the zeta-factor method, or ionization cross sections. The function iterates through quantification function until two successive iterations don’t change the final composition by a defined percentage criteria (0.5% by default).

Parameters

- **intensities** *(list of signal)* – the intensity for each X-ray lines.
- **method** *({'CL', 'zeta', '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** *({'atomic', 'weight'})* – The quantification returns the composition in ‘atomic’ percent by default, but can also return weight percent if specified.
- **absorption_correction** *(bool)* – Specify whether or not an absorption correction should be applied. ‘False’ by default so absorption will not be applied unless specified.
- **take_off_angle** *({'auto'})* – The angle between the sample surface and the vector along which X-rays travel to reach the centre of the detector.
- **thickness** *({'auto'})* – thickness in nm (can be a single value or have the same navigation dimension as the signal). NB: Must be specified for ‘CL’ method. For ‘zeta’ or ‘cross_section’ methods, first quantification step provides a mass_thickness internally during quantification.
- **convergence_criterion** *(The convergence criterium defined as the percentage)* – difference between 2 successive iterations. 0.5% by default.
- **navigation_mask** *(None or float or signal)* – The navigation locations marked as True are not used in the quantification. If float is given the vacuum_mask method is used to generate a mask with the float 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.

• `= {'auto'}` (*probe_area*) – This allows the user to specify the `probe_area` for interaction with the sample needed specifically for the `cross_section` method of quantification. When left as ‘auto’ the pixel area is used, calculated from the navigation axes information.

• **max_iterations** (*int*) – An upper limit to the number of calculations for absorption correction.

• **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 with absorption correction taken
  • into account based on the sample thickness estimate provided.
  • 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 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
HyperSpy Documentation, Release 1.6.0

- **azimuth_angle** *(float)* – In degree
- **elevation_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], dtype=bool)
```

**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

_alias_signal_types = ['TEM EELS']

_are_microscope_parameters_missing(ignore_parameters=[])
Check if the EELS parameters necessary to calculate the GOS are defined in metadata. If not, in interactive mode raises an UI item to fill the values. The ignore_parameters list can be to ignore parameters.

_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. Note that ROIs can be used in place of a tuple.

• **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.

```python
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 (bool, 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 (bool, 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 add the corresponding ionisation edges to the model.

• **GOS ({'hydrogenic' | 'Hartree-Slater'}, optional)** – The generalized oscillation strenght 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.

edges_at_energy (energy='interactive', width=10, only_major=False, order='closest', display=True, toolkit=None)

Show EELS edges according to an energy range selected from the spectrum or within a provided energy window

Parameters

• **energy ('interactive' or float)** – If it is ‘interactive’, a table with edges are shown and it depends on the energy range selected in the spectrum. If it is a float, a table with edges are shown and it depends on the energy window defined by energy +/- (width/2). The default is ‘interactive’.

• **width (float)** – Width of window, in eV, around energy in which to find nearby energies, i.e. a value of 10 eV (the default) means to search +/- 5 eV. The default is 10.

• **only_major (bool)** – Whether to show only the major edges. The default is False.

• **order (str)** – Sort the edges, if ‘closest’, return in the order of energy difference, if ‘ascending’, return in ascending order, similarly for ‘descending’. The default is ‘closest’.

Returns

• An interactive widget if energy is ‘interactive’, or a html-format

• table or ASCII table, depends on the environment.

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 \( I_0 \) – The elastic scattering intensity.

Return type \( \text{Signal1D} \)

See also:

\( \text{estimate_elastic_scattering_threshold}() \)

\( \text{estimate_elastic_scattering_threshold} \) (\( \text{window}=10.0, \ to\text{l}=\text{None}, \ \text{window_length}=5, \ \text{polynomial_order}=3, \ \text{start}=1.0 \) )

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 \( \text{start} + \text{window} \) 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

- \( \text{window} \) (\{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 \( (0, \text{window}] \) 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.
- \( \text{tol} \) (\{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.
- \( \text{window_length} \) (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.
- \( \text{polynomial_order} \) (int) – Savitzky-Golay filter polynomial order.
- \( \text{start} \) (float) – Position from the zero-loss peak centre from where to start looking for the inflexion point.

Returns \( \text{threshold} \) – A \( \text{Signal1D} \) 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 \( \text{nan} \).

Return type \( \text{Signal1D} \)

See also:

\( \text{estimate_elastic_scattering_intensity}(), \ \text{align_zero_loss_peak}(), \ \text{find_peaks1D_ohaver}(), \ \text{fourier_ratio_deconvolution}.() \)

Notes

The main purpose of this method is to be used as input for \( \text{estimate_elastic_scattering_intensity} \). Indeed, for currently achievable energy resolutions, there is not such a thing as a elastic scattering threshold. Therefore, please be aware of the limitations of this method when using it.

\( \text{estimate_thickness} \) (\( \text{threshold}=\text{None}, \ \text{zlp}=\text{None}, \ \text{density}=\text{None}, \ \text{mean_free_path}=\text{None} \) )

Estimates the thickness (relative and absolute) 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. To obtain the thickness relative to the mean free path don’t set the \text{density} and the \text{mean_free_path}.

Parameters
• **threshold**({BaseSignal, float, optional}) – If the zero-loss-peak is not provided, use this energy threshold to roughly estimate its intensity by truncation. If the threshold is constant across the dataset use a float. Otherwise, provide a signal of the same dimension as the input spectrum navigation space containing the threshold value in the energy units.

• **zlp** (BaseSignal, optional) – If not None the zero-loss peak intensity is calculated from the ZLP spectrum supplied by integration.

• **mean_free_path** (float, optional) – The mean free path of the material in nanometers. If not provided, the thickness is given relative to the mean free path.

• **density** (float, optional) – The density of the material in g/cm**3**. This is used to estimate the mean free path when the mean free path is not known and to perform the angular corrections.

Returns s – The thickness relative to the MFP. It returns a Signal1D, Signal2D or a BaseSignal, depending on the current navigation dimensions.

Return type BaseSignal

Notes


**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* (Optional) – 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

`get_complementary_edges` (*edges*, *only_major=False*)
Get other edges of the same element present within the energy range of the axis

**Parameters**
- *edges* (iterable) – A sequence of strings contains edges in the format of element_subshell for EELS. For example, ['Fe_L2', 'O_K']
- *only_major* (bool) – Whether to show only the major edges. The default is False.

**Returns**
- *complmt_edges* – A list containing all the complementary edges of the same element present within the energy range of the axis

**Return type** list

`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\(^1\). 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 iterations > 1.)

Raises

• ValuerError – 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\(^1\) with some minor differences:

• The wrap-around problem when computing the ffts is workarounded by padding the signal instead of substracting the reflected tail.

plot \((\text{plot\_edges}=False, \text{only\_edges}='Major', 'Minor', **kwargs)\)

Plot the EELS spectrum. Markers indicating the position of the EELS edges can be added.

Parameters

• plot_edges \( ((\text{False, True, list of string or string})\) – If True, draws on s.metadata.Sample.elements for edges. Alternatively, provide a string of a single edge, or an iterable containing a list of valid elements, EELS families or edges. For example, an element should be ‘Zr’, an element edge family should be ‘Zr_L’ or an EELS edge ‘Zr_L3’.
• only_edges \( (\text{tuple of string})\) – Either ‘Major’ or ‘Minor’. Defaults to both.
• kwargs – The extra keyword arguments for plot()

plot_edges_label \((\text{edges}, \text{vertical\_line\_marker}=\text{None}, \text{text\_marker}=\text{None})\)

Put the EELS edge label (vertical line segment and text box) on the signal

Parameters

• edges \( (\text{dictionary})\) – A dictionary with the labels as keys and their energies as values. For example, \{‘Fe_L2’: 721.0, ‘O_K’: 532.0\}
• vertical_line_marker \( (\text{list})\) – A list contains HyperSpy’s vertical line segment marker, if None, determine from the given edges
• text_marker \( (\text{list})\) – A list contains HyperSpy’s text box marker, if None, determine from the given edges

Raises ValueError – If the size of edges, vertical_line_marker and text_marker do not match.

power_law_extrapolation \((\text{window\_size}=20, \text{extrapolation\_size}=1024, \text{add\_noise}=False, \text{fix\_neg\_r}=False)\)

Extrapolate the spectrum to the right using a powerlaw

Parameters

• window_size \( (\text{int})\) – The number of channels from the right side of the spectrum that are used to estimate the power law parameters.
• extrapolation_size \( (\text{int})\) – Size of the extrapolation in number of channels
• add_noise \( (\text{bool})\) – If True, add poissonian noise to the extrapolated spectrum.
• fix_neg_r \( (\text{bool})\) – 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.

static print_edges_near_energy(energy=None, width=10, only_major=False, order='closest', edges=None)

Find and print a table of edges near a given energy that are within the given energy window.

Parameters

- energy (float) – Energy to search, in eV
- width (float) – Width of window, in eV, around energy in which to find nearby energies, i.e. a value of 10 eV (the default) means to search +/- 5 eV. The default is 10.
- only_major (bool) – Whether to show only the major edges. The default is False.
- order (str) – Sort the edges, if ‘closest’, return in the order of energy difference, if ‘ascending’, return in ascending order, similarly for ‘descending’. The default is ‘closest’.
- edges (iterable) – A sequence of edges, if provided, it overrides energy, width, only_major and order.

Returns

- A PrettyText object where its representation is ASCII in terminal and html-formatted in Jupyter notebook

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)
<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_EELS_edges_markers(EELS_edges)`

`richardson_lucy_deconvolution(psf, iterations=15, mask=None, show_progressbar=None, parallel=None, max_workers=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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.
- `max_workers (None or int)` – Maximum number of threads used when parallel=True. If None, defaults to `min(32, os.cpu_count())`.

Notes


`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.

If not all of them are defined, in interactive mode raises an UI item to fill the values

- `beam_energy: float` The energy of the electron beam in keV
- `convergence_angle` [float] The microscope convergence 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.

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.

mapping = {'Acquisition_instrument.TEM.Detector.EELS.collection_angle': 'collection_angle',
            'Acquisition_instrument.TEM.beam_energy': 'beam_energy',
            'Acquisition_instrument.TEM.convergence_angle': 'convergence_angle'}

class hyperspy._signals.eels.LazyEELSSpectrum (*args, **kwargs)
Bases: hyperspy._signals.eels.EELSSpectrum, hyperspy._signals.signal1d.LazySignal1D

hyperspy._signals.hologram_image module

class hyperspy._signals.hologram_image.HologramImage (*args, **kw)
Bases: hyperspy._signals.signal2d.Signal2D

Image subclass for holograms acquired via off-axis electron holography.

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
     store 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.

__signal_type = 'hologram'

estimate_sideband_position (ap_cb_radius=None, sb='lower', high Cf=True,
                           show_progressbar=False, parallel=None, max_workers=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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

- **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**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, max_workers=None)*

Estimates the size of the sideband and returns its size.

**Parameters**

- **sb_position** *(BaseSignal)* – The sideband position (y, x), referred to the non-shifted FFT.

- **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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

- **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**Returns** `sb_size` – Sideband size referred to the unshifted FFT.

**Return type** `Signal1D`

**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, store_parameters=True, show_progressbar=False, parallel=None, max_workers=None)

Reconstruct electron holograms. Operates on multidimensional hyperspy signals. There are several usage schemes:

- Reconstruct 1d or Nd hologram without reference
- Reconstruct 1d or Nd hologram using single reference hologram
- 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

- **reference**(ndarray, Signal2D, None) – Vacuum reference hologram.
- **sb_size**(float, ndarray, 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, 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, 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**(bool) – Shows details of the reconstruction (i.e. SB selection).
- **store_parameters**(bool) – Store reconstruction parameters in metadata
- **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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.
- **max_workers**(None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

Returns *wave* – Reconstructed electron wave. By default object wave is divided by reference wave.

Return type *ComplexSignal2D*
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)
>>> 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**

```python
>>> 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, max_workers=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, 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 the interference pattern do not cover full field of view.
  - ‘statistical’ : fringe contrast is estimated by dividing the standard deviation by the mean of the hologram intensity in real space. This algorithm relies on regularly spaced fringes and covering the entire field of view.

(Defaults: ‘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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

• **max_workers**(None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**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}
```

class hyperspy._signals.hologram_image.LazyHologramImage(*args, **kw)
Bases: hyperspy._signals.lazy.LazySignal, hyperspy._signals.hologram_image.HologramImage

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.

  _lazy = True
hyperSpy Documentation, Release 1.6.0

hyperSpy._signals.hologram_image._estimate_fringe_contrast_statistical(holo)
Estimates average fringe contrast of a hologram using statistical definition: \( V = \frac{\text{STD}}{\text{MEAN}} \).

**Parameters**

holo_data (ndarray) – The data of the hologram.

**Returns**

Return type: Fringe contrast as a float

hyperSpy._signals.hologram_image._first_nav_pixel_data(s)

hyperSpy._signals.hologram_image._parse_sb_position(s, reference, sb_position, sb, high_cf, parallel)

hyperSpy._signals.hologram_image._parse_sb_size(s, reference, sb_position, sb_size, parallel)

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.

_hyper_function_on_data_and_remove_axis_(function, axes, out=None, rechunk=True)

_block_iterator_(flat_signal=True, get=<function get>, navigation_mask=None, signal_mask=None)

A function that allows iterating lazy signal data by blocks, defining the dask.Array.

**Parameters**

• flat_signal (bool) – returns each block flattened, such that the shape (for the particular block) is (navigation_size, signal_size), with optionally masked elements missing. If false, returns the equivalent of s.inav[...,blocks].data, where masked elements are set to np.nan or 0.
• get (dask scheduler) – the dask scheduler to use for computations; default dask.threaded.get
• navigation_mask ((BaseSignal, numpy array, dask array)) – The navigation locations marked as True are not returned (flat) or set to NaN or 0.
• signal_mask ((BaseSignal, numpy array, dask array)) – The signal locations marked as True are not returned (flat) or set to NaN or 0.
_calculate_summary_statistics(rechunk=True)

static _estimate_poissonian_noise_variance(dc, gain_factor, gain_offset, correlation_factor)

_get_dask_chunks(axis=None, dtype=None)

Returns dask chunks.

Aims:

- Have at least one signal (or specified axis) in a single chunk, or as many as fit in memory

Parameters

- **axis** ([int, string, None, axis, tuple]) – If axis is None (default), returns chunks for current data shape so that at least one signal is in the chunk. If an axis is specified, only that particular axis is guaranteed to be “not sliced”.

- **dtype** ([string, np.dtype]) – The dtype of target chunks.

Returns

Return type: Tuple of tuples, dask chunks

_iterate_signal()

Iterates over the signal data.

It is faster than using the signal iterator.

_lazy = True

_lazy_data(axis=None, rechunk=True, dtype=None)

Return the data as a dask array, rechunked if necessary.

Parameters

- **axis** ([None, DataAxis or tuple of data axes]) – The data axis that must not be broken into chunks when rechunk is True. If None, it defaults to the current signal axes.

- **rechunk** (bool, "dask_auto") – If True, it rechunks the data if necessary making sure that the axes in axis are not split into chunks. If False it does not rechunk at least the data is not a dask array, in which case it chunks as if rechunk was True. If “dask_auto”, rechunk if necessary using dask’s automatic chunk guessing.

_make_lazy(axis=None, rechunk=False, dtype=None)

_make_sure_data_is_contiguous()

_map_all(function, inplace=True, **kwargs)

The function has to have either ‘axis’ or ‘axes’ keyword argument, and hence support operating on the full dataset efficiently.

Replaced for lazy signals

_map_iterate(function, iterating_kwargs=(), show_progressbar=None, parallel=None, max_workers=None, ragged=None, inplace=True, **kwargs)

Iterates the signal navigation space applying the function.

Parameters

- **function** (function) – the function to apply
• **iterating_kwargs** (*tuple (of tuples)*) – A tuple with structure ((‘key1’,
value1), (‘key2’, value2), ..) where the key-value pairs will be passed as kwargs for the
function to be mapped, and the values will be iterated together with the signal navigation.

• **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 multi-
threading. If None, the default from the preferences settings is used. The number of
threads is controlled by the max_workers argument.

• **max_workers** (*None or int*) – Maximum number of threads used when
parallel=True. If None, defaults to min(32, os.cpu_count()).

• **inplace** (*bool, default True*) – If True, the data is replaced by the result. Oth-
otherwise a new signal with the results is returned.

• **ragged** (*None or bool, default None*) – Indicates if results for each naviga-
tion pixel are of identical shape (and/or numpy arrays to begin with). If None, an appro-
propriate choice is made while processing. Note: None is not allowed for Lazy signals!

• ****kwargs** (*dict*) – Additional keyword arguments passed to function

**Notes**

This method is replaced for lazy signals.

**Examples**

Pass a larger array of different shape

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((20,1)))
>>> def func(data, value=0):
...   return data + value
>>> # pay attention that it's a tuple of tuples - need commas
>>> s._map_iterate(func,
...   iterating_kwargs=('value',
...   np.random.rand(5,400).flat),))
>>> s.data.T
array([[ 0.82869603, 1.04961735, 2.21513949, 3.61329091,
        4.2481755 , 5.81184375, 6.47696867, 7.07682618,
        8.16850697, 9.37771809, 10.42794054, 11.24362699,
       12.11434077, 13.98654036, 14.72864184, 15.30855499,
       16.96854373, 17.65077064, 18.64925703, 19.16901297]])
```

Storing function result to other signal (e.g. calculated shifts)

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((5,4)))
>>> def func(data):
# the original function
...   return data.sum()
>>> result = s._get_navigation_signal().T
>>> def wrapped(*args, data=None):
...   return func(data)
>>> result._map_iterate(wrapped,
...   iterating_kwargs=('data', s),))
>>> result.data
array([ 6., 22., 38., 54., 70.])
```
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)8 or rgb(a)16, and the navigation_dimension should be at least 2. After conversion, the signal_dimension becomes 2. The dtype of images with original dtype rgb(a)8 or rgb(a)16 can only be changed to uint8 or uint16, and the signal_dimension becomes 1.

- **rechunk** (bool) – Only has effect when operating on lazy signal. 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, print_info=True, **kwargs)

Perform Incremental (Batch) decomposition on the data.

The results are stored in self.learning_results.

Read more in the User Guide.

Parameters

- **normalize_poissonian_noise** (bool, default False) – If True, scale the signal to normalize Poissonian noise using the approach described in [KeenanKotula2004].


- **output_dimension** (int or None, default None) – Number of components to keep/calculate. If None, keep all (only valid for ‘SVD’ algorithm)
• **get** *(dask scheduler)* – the dask scheduler to use for computations; default `dask.threaded.get`

• **num_chunks** *(int or None, default None)* – the number of dask chunks to pass to the decomposition model. More chunks require more memory, but should run faster. Will be increased to contain at least `output_dimension` signals.

• **navigation_mask** *(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, default True)* – Reproject data on the learnt components (factors) after learning.

• **print_info** *(bool, default True)* – If True, print information about the decomposition being performed. In the case of `sklearn.decomposition` objects, this includes the values of all arguments of the chosen sklearn algorithm.

• ****kwargs** – passed to the partial_fit/fit functions.

### References

See also:

• **decomposition** *(for non-lazy signals)*

• `dask.array.linalg.svd()`

• `sklearn.decomposition.IncrementalPCA`

• **ORPCA**

• **ORNMF**

### 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.

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)
```

get_histogram(bins='fd', out=None, rechunk=True, **kwargs)

Return a histogram of the signal data.

More sophisticated algorithms for determining the bins can be used by passing a string as the `bins` argument. Other than the 'blocks' and 'knuth' methods, the available algorithms are the same as `numpy.histogram()`.

Note: The lazy version of the algorithm only supports "scott" and "fd" as a string argument for `bins`.

Parameters

- `bins` *(int or sequence of scalars or str, default "fd")* - If `bins` is an int, it defines the number of equal-width bins in the given range. If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If `bins` is a string from the list below, will use the method chosen to calculate the optimal bin width and consequently the number of bins (see Notes for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the 'auto' option is suggested. Weighted data is not supported for automated bin size selection.

- 'auto' - Maximum of the 'sturges' and 'fd' estimators. Provides good all around performance.
- 'fd' *(Freedman Diaconis Estimator)* - Robust (resilient to outliers) estimator that takes into account data variability and data size.
- 'doane' - An improved version of Sturges' estimator that works better with non-normal datasets.
- 'scott' - Less robust estimator that that takes into account data variability and data size.
- 'stone' - Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott’s rule.
- 'rice' - Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.
- 'sturges' - R’s default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.
- 'sqrt' - Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.
- 'knuth' - Knuth’s rule is a fixed-width, Bayesian approach to determining the optimal bin width of a histogram.
- 'blocks' - Determination of optimal adaptive-width histogram bins using the Bayesian Blocks algorithm.

- `range_bins` *(tuple or None, optional)* - the minimum and maximum range for the histogram. If `range_bins` is None, `(x.min(), x.max())` will be used.
• **max_num_bins** (*int*, *default 250*) – When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a MemoryError being raised by `numpy.histogram()`.

• **out** ([`BaseSignal` (or 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`
- `numpy.histogram()`
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)
<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)>
```
```python
>>> 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()
```
hyperspy._signals.lazy._reshuffle_mixed_blocks(array, ndim, sshape, nav_chunks)

Reshuffles dask block-shuffled array

Parameters

- **array** ([np.ndarray]) – the array to reshuffle
- **ndim** ([int]) – the number of navigation (shuffled) dimensions
- **sshape** ([tuple of ints]) – The shape

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}
```

**hyperspy._signals.signal1d module**

```
class hyperspy._signals.signal1d.LazySignal1D(*args, **kwargs)

Bases: hyperspy._signals.lazy.LazySignal, hyperspy._signals.signal1d.Signal1D

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.

_lazy = True
```

```
class hyperspy._signals.signal1d.Signal1D(*args, **kwargs)

Bases: hyperspy.signal.BaseSignal, hyperspy._signals.common_signal1d.CommonSignal1D

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 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.

_integrate_in_range_commandline (signal_range)

_remove_background_cli (signal_range, background_estimator, fast=True, zero_fill=False,
                   show_progressbar=None, model=None, return_model=False)
          See remove_background().

_signal_dimension = 1

_spikes_diagnosis (signal_mask=None, navigation_mask=None)

Plots a histogram to help in choosing the threshold for spikes removal.

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).

See also:

spikes_removal_tool()

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:**

*estimate_shift1D()*

**calibrate** (*display=True, toolkit=None*)

Calibrate the spectral dimension using a gui. It displays a window where the new calibration can be set by:

• setting the values of offset, units and scale directly

• or selecting a range by dragging the mouse on the spectrum figure and setting the new values for the given range limits

**Parameters**

• **display** (*bool*) – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

• **toolkit** (*str, iterable of 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, max_workers=None)*

Estimate the width of the highest intensity of peak of the spectra at a given fraction of its maximum.

It can be used with asymmetric peaks. For accurate results any background must be previously 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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.
- **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to `min(32, os.cpu_count())`.

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, max_workers=None)*

Estimate the shifts in the current signal axis using cross-correlation. This method can only estimate the shift by comparing unidimensional features that should not change the position in the signal axis. To decrease the memory usage, the time of computation and the accuracy of the results it is convenient to select the feature of interest providing sensible values for `start` and `end`. By default interpolation is used to obtain subpixel precision.

Parameters

- **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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

• **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**Returns**

• 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** *(xdim=None, slope_thresh=0, amp_thresh=None, subchannel=True, medfilt_radius=5, maxpeakn=30000, peakgroup=10, parallel=None, max_workers=None)*

Find positive peaks along a 1D Signal. It detects peaks by looking for downward zero-crossings in the first derivative that exceed ‘slope_thresh’.

‘slope_thresh’ and ‘amp_thresh’, control sensitivity: higher values will neglect broad peaks (slope) and smaller features (amp), respectively.

*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, default True) – default is set to True.

• **parallel**(None or bool) – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

• **max_workers**(None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

**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. Note that ROIs can be used in place of a tuple.

**Returns** integrated_spectrum

**Return type** BaseSignal subclass

See also:

integrate_simpson()

### Examples

Using the GUI

```python
>>> s = hs.signals.Signal1D(range(1000))
>>> s.integrate_in_range()
```

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, show_progressbar=None, parallel=None, max_workers=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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.

- **max_workers**(None or int) - Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

- ****kwargs – All extra keyword arguments are passed to scipy.interpolate.interp1d(). See the function documentation for details.

**Raises** SignalDimensionError – If the signal dimension is not 1.

**plot**(navigator='auto', plot_markers=True, autoscale='v', norm='auto', axes_manager=None, navigator_kwds={}, **kwargs)

Plot the signal at the current coordinates.
For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

- **navigator** (str, None, or BaseSignal (or subclass)) –
  - 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 an 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.
- **navigator_kwds** (dict) – Only for image navigator, additional keyword arguments for matplotlib.pyplot.imshow().
- **norm** (str, optional) – The function used to normalize the data prior to plotting. Allowable strings are: 'auto', 'linear', 'log'. (default value is 'auto').
'auto', intensity is plotted on a linear scale except when power_spectrum=True
(only for complex signals).

- **autoscale** *(str)* – The string must contain any combination of the ‘x’ and ‘v’ characters. If ‘x’ or ‘v’ (for values) are in the string, the corresponding horizontal or vertical axis limits are set to their maxima and the axis limits will reset when the data or the navigation indices are changed. Default is ‘v’.

```python
remove_background(signal_range='interactive', background_type='Power law', polynomial_order=2, fast=True, zero_fill=False, plot_remainder=True, show_progressbar=None, return_model=False, display=True, toolkit=None)
```

Remove the background, either in place using a GUI or returned as a new spectrum using the command line. The fast option is not accurate for most background types - except Gaussian, Offset and Power law - but it is useful to estimate the initial fitting parameters before performing a full fit.

**Parameters**

- **signal_range** *("interactive", tuple of 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: Doniach, Gaussian, Lorentzian, Offset, Polynomial, PowerLaw, Exponential, SkewNormal, SplitVoigt, Voigt. If Polynomial is used, the polynomial order can be specified.

- **polynomial_order** *(int, default 2)* – Specify the polynomial order if a Polynomial background is used.

- **fast** *(bool)* – If True, perform an approximative estimation of the parameters. If False, the signal is fitted using non-linear least squares afterwards. This is slower compared to the estimation but often more accurate.

- **zero_fill** *(bool)* – If True, all spectral channels lower than the lower bound of the fitting range will be set to zero (this is the default behavior of Gatan’s DigitalMicrograph). Setting this value to False allows for inspection of the quality of background fit throughout the pre-fitting region.

- **plot_remainder** *(bool)* – If True, add a (green) line previewing the remainder signal after background removal. This preview is obtained from a Fast calculation so the result may be different if a NLLS calculation is finally performed.

- **return_model** *(bool)* – If True, the background model is returned. The chi^2 can be obtained from this model using chisqd().

- **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.

**Returns** If signal_range is not ‘interactive’, the signal with background substracted is returned. If return_model is True, returns the background model, otherwise, the GUI widget dictionary is returned if display=False - see the display parameter documentation.

**Return type** *(None, signal, background_model or (signal, background_model))
Examples

Using GUI, replaces spectrum s

```python
>>> s = hs.signals.Signal1D(range(1000))
>>> s.remove_background()
```

Using command line, returns a Signal1D:

```python
>>> s.remove_background(signal_range=(400,450),
                      background_type='PowerLaw')
<Signal1D, title: , dimensions: (|1000)>
```

Using a full model to fit the background:

```python
>>> s.remove_background(signal_range=(400,450), fast=False)
<Signal1D, title: , dimensions: (|1000)>
```

Returns background substracted and the model:

```python
>>> s.remove_background(signal_range=(400,450),
                      fast=False, return_model=True)
(<Signal1D, title: , dimensions: (|1000)>, <Model1D>)
```

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, max_workers=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'`, `'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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.
- `max_workers (None or int)` – Maximum number of threads used when `parallel=True`. If None, defaults to `min(32, os.cpu_count())`.

Raises `SignalDimensionError` – If the signal dimension is not 1.
smooth_lowess(smoothing_parameter=None, number_of_iterations=None, show_progressbar=None, parallel=None, max_workers=None, display=True, toolkit=None)

Lowess data smoothing in place. If smoothing_parameter or number_of_iterations are None the method is run in interactive mode.

Parameters

- smoothing_parameter(float or None) – Between 0 and 1. The fraction of the data used when estimating each y-value.
- number_of_iterations(int or None) – The number of residual-based reweightings to perform.
- show_progressbar(None or bool) – If True, display a progress bar. If None, the default from the preferences settings is used.
- parallel(None or bool) – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.
- max_workers(None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).
- 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(polyomial_order=None, window_length=None, differential_order=0, parallel=None, max_workers=None, display=True, toolkit=None)

Apply a Savitzky-Golay filter to the data in place. If polynomial_order or window_length or differential_order are None the method is run in interactive mode.

Parameters

- polynomial_order(int, optional) – The order of the polynomial used to fit the samples. polyorder must be less than window_length.
- window_length(int, optional) – The length of the filter window (i.e. the number of coefficients). window_length must be a positive odd integer.
- differential_order(int, optional) – The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.
• **parallel** *(None or bool)* – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.

• **max_workers** *(None or int)* – Maximum number of threads used when `parallel=True`. If None, defaults to `min(32, os.cpu_count())`.

• **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`.

```python
smooth_tv(smoothing_parameter=None, show_progressbar=None, parallel=None, max_workers=None, display=True, toolkit=None)
```

Total variation data smoothing in place.

**Parameters**

• **smoothing_parameter** *(float or None)* – Denoising weight relative to L2 minimization. If None the method is run in interactive mode.

• **show_progressbar** *(None or bool)* – If True, display a progress bar. If None, the default from the preferences settings is used.

• **parallel** *(None or bool)* – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.

• **max_workers** *(None or int)* – Maximum number of threads used when `parallel=True`. If None, defaults to `min(32, os.cpu_count())`.

• **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.

```python
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.

See also:

`_spikes_diagnosis()`

```python
hyperspy._signals.signal1d._estimate_shift1D(data, **kwargs)
```

```python
hyperspy._signals.signal1d._shift1D(data, **kwargs)
```

```python
hyperspy._signals.signal1d.find_peaks_ohaver(y, x=None, slope_thresh=0.0, amp_thresh=None, medfilt_radius=5, maxpeakn=30000, peakgroup=10, subchannel=True)
```

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.

**Returns**

- **P** – contains fields: ‘position’, ‘width’, and ‘height’ for each peak.

**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


`hyperspy._signals.signal1d.interpolate1D(number_of_interpolation_points, data)`

`hyperspy._signals.signal2d` module

class `hyperspy._signals.signal2d.LazySignal2D(*args, **kwargs)`

Bases: `hyperspy._signals.lazy.LazySignal, hyperspy._signals.signal2d.Signal2D`

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.

_lazy = True

class `hyperspy._signals.signal2d.Signal2D(*args, **kw)`

Bases: `hyperspy.signal.BaseSignal, hyperspy._signals.common_signal2d.CommonSignal2D`

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.

_lazy = False
_signal_dimension = 2

add_ramp (ramp_x, ramp_y, offset=0)
Add a linear ramp to the signal.

Parameters

- **ramp_x** *(float)* – Slope of the ramp in x-direction.
- **ramp_y** *(float)* – Slope of the ramp in y-direction.
- **offset** *(float, optional)* – Offset of the ramp at the signal fulcrum.

Notes

The fulcrum of the linear ramp is at the origin and the slopes are given in units of the axis with the according scale taken into account. Both are available via the `axes_manager` of the signal.

align2D (crop=True, fill_value=nan, shifts=None, expand=False, interpolation_order=1, show_progressbar=None, parallel=None, max_workers=None, **kwargs)
Align the images in-place using `scipy.ndimage.shift()`.

The images can be aligned using either user-provided shifts or by first estimating the shifts.

See `estimate_shift2D()` for more details on estimating image shifts.

Parameters

- **crop** *(bool)* – If True, the data will be cropped not to include regions with missing data
- **fill_value** *(int, float, 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. Over-rides crop.
- **interpolation_order** *(int, default 1.)* – The order of the spline interpolation. Default is 1, linear interpolation.
- **show_progressbar** *(None or bool)* – If True, display a progress bar. If None, the default from the preferences settings is used.
- **parallel** *(None or bool)* – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.
- **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).
- ****kwargs **– Keyword arguments passed to `estimate_shift2D()`

Returns **shifts** – The estimated shifts are returned only if `shifts` is None

Return type **np.array**

See also:

- `estimate_shift2D()`

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, None\}, bottom=\{None, None\}, left=\{None, None\}, right=\{None, 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.

See also:
crop()
estimate_shift2D (reference=\{'current', 'cascade', 'stat'\}, correlation_threshold=\{None, 'auto', float\}, chunk_size=30, roi=\{None, None\}, normalize_corr=False, sobel=True, medfilter=True, hanning=True, plot=False, dtype='float', show_progressbar=None, sub_pixel_factor=1)
Estimate the shifts in an image using phase correlation.

This method can only estimate the shift by comparing bi-dimensional features that should not change position between frames. To decrease the memory usage, the time of computation and the accuracy of the results it is convenient to select a region of interest by setting the roi argument.

Parameters

• reference (\{'current', 'cascade', 'stat'\}) – If ‘current’ (default) the image at the current coordinates is taken as reference. If ‘cascade’ each image is aligned with the previous one. If ‘stat’ the translation of every image with all the rest is estimated and by performing statistical analysis on the result the translation is estimated.

• correlation_threshold (\{None, 'auto', float\}) – This parameter is only relevant when reference=‘stat’. If float, the shift estimations with a maximum correlation value lower than the given value are not used to compute the estimated shifts. If ‘auto’ the threshold is calculated automatically as the minimum maximum correlation value of the automatically selected reference image.

• chunk_size (\{None, 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). Note that ROIs can be used in place of a tuple.

• normalize_corr (bool, default False) – If True, use phase correlation to align the images, otherwise use cross correlation.

• sobel (bool, default True) – Apply a Sobel filter for edge enhancement

• medfilter (bool, default True) – Apply a median filter for noise reduction

• hanning (bool, default True) – Apply a 2D hanning filter

• plot (bool or '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**
| shifts | List of estimated shifts |

**Return type**
| list of array |

**Notes**

The statistical analysis approach to the translation estimation when using reference='stat' roughly follows [Schaffer2004]. If you use it please cite their article.

**References**

See also:

- **align2D()**

**find_peaks** *(method='local_max', interactive=True, current_index=False, show_progressbar=None, parallel=None, max_workers=None, display=True, toolkit=None, **kwargs)*

Find peaks in a 2D signal.

Function to locate the positive peaks in an image using various, user specified, methods. Returns a structured array containing the peak positions.

**Parameters**

- **method** *(str)* – Select peak finding algorithm to implement. Available methods are:
  - 'local_max' - simple local maximum search using the `skimage.feature.peak_local_max()` function
  - 'max' - simple local maximum search using the `find_peaks_max()`.
  - 'minmax' - finds peaks by comparing maximum filter results with minimum filter, calculates centers of mass. See the `find_peaks_minmax()` function.
  - 'zaefferer' - based on gradient thresholding and refinement by local region of interest optimisation. See the `find_peaks_zaefferer()` function.
  - 'stat' - based on statistical refinement and difference with respect to mean intensity. See the `find_peaks_stat()` function.
  - 'laplacian_of_gaussian' - a blob finder using the laplacian of Gaussian matrices approach. See the `find_peaks_log()` function.
  - 'difference_of_gaussian' - a blob finder using the difference of Gaussian matrices approach. See the `find_peaks_log()` function.
  - 'template_matching' - A cross correlation peakfinder. This method requires providing a template with the `template` parameter, which is used as reference pattern to perform the template matching to the signal. It uses the `skimage.feature.match_template()` function and the peaks position are obtained by using `minmax` method on the template matching result.
• **interactive** (bool) – If True, the method parameter can be adjusted interactively. If False, the results will be returned.
• **current_index** (bool) – If True, the computation will be performed for the current index.
• **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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.
• max_workers (None or int) – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).
• **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.
• **kwargs** (dict) – Keywords parameters associated with above methods, see the documentation of each method for more details.

Notes

As a convenience, the ‘local_max’ method accepts the ‘distance’ and ‘threshold’ argument, which will be map to the ‘min_distance’ and ‘threshold_abs’ of the skimage.feature.peak_local_max() function.

Returns peaks – Array of shape _navigation_shape_in_array in which each cell contains an array with dimensions (npeaks, 2) that contains the x, y pixel coordinates of peaks found in each image sorted first along y and then along x.

Return type BaseSignal or numpy.ndarray if current_index=True

plot (navigator='auto', plot_markers=True, autoscale='v', saturated_pixels=None, norm='auto', vmin=None, vmax=None, gamma=1.0, linthresh=0.01, linscale=0.1, scalebar=True, scalebar_color='white', axes_ticks=None, axes_off=False, axes_manager=None, no_nans=False, colormap=True, centre_colormap='auto', min_aspect=0.1, navigator_kwds={}, **kwargs)

Plot the signal at the current coordinates.

For multidimensional datasets an optional figure, the “navigator”, with a cursor to navigate that data is raised. In any case it is possible to navigate the data using the sliders. Currently only signals with signal_dimension equal to 0, 1 and 2 can be plotted.

Parameters

• **navigator** (str, None, or BaseSignal (or subclass)) –

• 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.

• `navigator_kwds` (`dict`) – Only for image navigator, additional keyword arguments for `matplotlib.pyplot.imshow()`.

• `colorbar` (`bool`, optional) – If true, a colorbar is plotted for non-RGB images.

• `autoscale` (`str`) – The string must contain any combination of the ‘x’, ‘y’ and ‘v’ characters. If ‘x’ or ‘y’ are in the string, the corresponding axis limits are set to cover the full range of the data at a given position. If ‘v’ (for values) is in the string, the contrast of the image will be set automatically according to `vmin` and `vmax` when the data or navigation indices change. Default is ‘v’.

• `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.

• `deprecated`: (.) – 1.6.0: `saturated_pixels` will be removed in HyperSpy 2.0.0, it is replaced by `vmin`, `vmax` and `autoscale`.

• `norm` ("auto", "linear", "power", "log", "symlog" or a subclass of) – `matplotlib.colors.Normalize` Set the norm of the
image to display. If “auto”, a linear scale is used except if when `power_spectrum=True` in

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmax (vmin,)</code></td>
<td><code>vmin</code> and <code>vmax</code> are used to normalise the displayed data. It can be a</td>
</tr>
<tr>
<td></td>
<td>float or a string. If string, it should be formatted as ‘xth’, where ‘x’</td>
</tr>
<tr>
<td></td>
<td>must be an float in the [0, 100] range. ‘x’ is used to compute the x-th</td>
</tr>
<tr>
<td></td>
<td>percentile of the data. See <code>numpy.percentile()</code> for more information.</td>
</tr>
<tr>
<td><code>gamma (float)</code></td>
<td>Parameter used in the power-law normalisation when the parameter</td>
</tr>
<tr>
<td></td>
<td><code>norm=“power”</code>. Read <code>matplotlib.colors.PowerNorm</code> for more details.</td>
</tr>
<tr>
<td></td>
<td>Default value is 1.0.</td>
</tr>
<tr>
<td><code>linthresh (float)</code></td>
<td>When used with <code>norm=“symlog”</code>, define the range within which the</td>
</tr>
<tr>
<td></td>
<td>plot is linear (to avoid having the plot go to infinity around zero).</td>
</tr>
<tr>
<td></td>
<td>Default value is 0.01.</td>
</tr>
<tr>
<td><code>linscale (float)</code></td>
<td>This allows the linear range (-<code>linthresh</code> to <code>linthresh</code>) to be stretched</td>
</tr>
<tr>
<td></td>
<td>relative to the logarithmic range. Its value is the number of powers of</td>
</tr>
<tr>
<td></td>
<td>base to use for each half of the linear range. See <code>matplotlib.colors.SymLogNorm</code> for more details. Default value is 0.1.</td>
</tr>
<tr>
<td><code>scalebar (bool, optional)</code></td>
<td>If True and the units and scale of the x and y axes are the same a scale bar is plotted.</td>
</tr>
<tr>
<td><code>scalebar_color (str, optional)</code></td>
<td>A valid MPL color string; will be used as the scalebar color.</td>
</tr>
<tr>
<td><code>axes_ticks ((None, bool), optional)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>axes_off ((bool))</code></td>
<td>Default is False.</td>
</tr>
<tr>
<td><code>no_nans (bool, optional)</code></td>
<td>If True, set nans to zero for plotting.</td>
</tr>
<tr>
<td><code>centre_colormap (&quot;auto&quot;, True, False)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>min_aspect (float)</code></td>
<td>Set the minimum aspect ratio of the image and the figure. To keep the image in the aspect limit the pixels are made rectangular.</td>
</tr>
</tbody>
</table>
| `**kwargs` | Only when plotting an image: additional (optional) keyword arguments for `matplotlib.pyplot.imshow()`.

**hyperspy._signals.signal2d.estimate_image_shift**

```python
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**

- `ref (2D numpy.ndarry)` – 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 or 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’ or ‘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**0**. If you use it please cite their article.

**References**

**hyperspy._signals.signal2d.fft_correlation** *(in1, in2, normalize=False, real_only=False)*

Correlation of two N-dimensional arrays using FFT.

Adapted from scipy’s fftconvolve.

**Parameters**

• **in2** *(in1,) – Input arrays to convolve.*

• **normalize** *(bool, default False) – If True performs phase correlation.*

• **real_only** *(bool, default False) – If True, and in1 and in2 are real-valued inputs, uses rfft instead of fft for approx. 2x speed-up.*

**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)*

---

**triu_indices_minus_diag**

Returns the indices for the upper-triangle of an \((n, n)\) array excluding its diagonal

**Parameters**

- \(n\) \((\text{int})\) – 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.

**get_atomic_resolution_tem_signal2d**

Get an artificial atomic resolution TEM Signal2D.

**Returns**

**Return type** `Signal2D`

**Example**

```python
>>> s = hs.datasets.artificial_data.get_atomic_resolution_tem_signal2d()
>>> s.plot()
```

**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.

**Parameters**

- `add_powerlaw` \((\text{bool})\) – If True, adds a powerlaw background to the spectrum. Default is False.
- `add_noise` \((\text{bool})\) – If True, add noise to the signal. See note to seed the noise to generate reproducible noise.
- `random_state` \((\text{None or int or RandomState instance, default None})\) – Random seed used to generate the data.

**Returns**

**Return type** `EELSSpectrum`
Example

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

See also:

`get_low_loss_eels_line_scan_signal()`, `get_core_loss_eels_model()`

Get an artificial core loss electron energy loss model.

Similar to a Mn-L32 edge from a perovskite oxide.

**Parameters**

- `add_powerlaw` *(bool)* – If True, adds a powerlaw background to the spectrum. Default is False.
- `add_noise` *(bool)* – If True, add noise to the signal. See note to seed the noise to generate reproducible noise.
- `random_state` *(None or int or RandomState instance, default None)* – Random seed used to generate the data.

**Returns**

Return type *EELSModel*

Example

```python
>>> import hs.datasets.artifical_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_core_loss_eels_signal()`
- **add_noise** *(bool)* – If True, add noise to the signal. See note to seed the noise to generate reproducible noise.

- **random_state** *(None or int or RandomState instance, default None)* – Random seed used to generate the data.

**Returns**

**Return type** *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
>>> s1 = ad.get_core_loss_eels_signal(random_state=10)
>>> s2 = ad.get_core_loss_eels_signal(random_state=10)
>>> (s1.data == s2.data).all()
True
```

**See also:**

`get_core_loss_eels_line_scan_signal()`, `get_low_loss_eels_line_scan_signal()`, `get_core_loss_eels_model()`

### hyperspy.datasets.artificial_data.get_low_loss_eels_line_scan_signal*(add_noise=True, random_state=None)*

Get an artificial low loss electron energy loss line scan spectrum.

The zero loss peak is offset by 4.1 eV.

**Parameters**

- **add_noise** *(bool)* – If True, add noise to the signal. See note to seed the noise to generate reproducible noise.

- **random_state** *(None or int or RandomState instance, default None)* – Random seed used to generate the data.

**Returns**

**Return type** *EELSSpectrum*
Example

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

See also:

- `artificial_low_loss_line_scan_signal()` : EELSSpectrum

**hyperspy.datasets.artificial_data.get_low_loss_eels_signal** *(add_noise=True, random_state=None)*

Get an artificial low loss electron energy loss spectrum.

The zero loss peak is offset by 4.1 eV.

**Parameters**

- `add_noise` *(bool)* – If True, add noise to the signal. See note to seed the noise to generate reproducible noise.

- `random_state` *(None or int or RandomState instance, default None)* – Random seed used to generate the data.

**Returns**

**Return type** EELSSpectrum

Returns `artificial_low_loss_signal`

**Return type** EELSSpectrum

Example

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

See also:

- `get_core_loss_eels_signal()`, `get_core_loss_eels_model()`,
- `get_low_loss_eels_line_scan_signal()`, `get_core_loss_eels_line_scan_signal()`

**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.model module

Common docstring snippets for model.

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.horizonal_line module

```python
class hyperspy.drawing._markers.horizonal_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)
```

```python
def _plot_marker()
update()
```

**hyperspy.drawing._markers.horizontal_line_segment module**

```python
class 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)
```

```python
def _plot_marker()
```
hyperspy.drawing._markers.line_segment module

class hyperspy.drawing._markers.line_segment.LineSegment(x1, y1, x2, y2, **kwargs)

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)
```

```python
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)
```

_plot_marker()
_update_segment()
update()}
hyperspy.drawing._markers.point module

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
c
>>> 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)
```

_plot_marker()

update()
hyperspy.drawing._markers.rectangle module

class hyperspy.drawing._markers.rectangle.Rectangle(x1, y1, x2, y2, **kwargs)

Bases: hyperspy.drawing.marker.MarkerBase

Rectangle marker that can be added to the signal figure

Parameters

- **x1 (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.

- **y1 (array or float)** – The position of the up left corner of the rectangle in y. see x1 arguments

- **x2 (array or float)** – The position of the down right corner of the rectangle in x. see x1 arguments

- **y2 (array or float)** – The position of the down right of the rectangle in y. see x1 arguments

- **kwargs** – 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

- **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.

- **y (array or float)** – The position of the text in y. see x arguments

- **text (array or str)** – The text. see x arguments
• **kwargs** – 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)
```

**hyperspy.drawing._markers.vertical_line module**

```python
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((100, 100)))
>>> m = hs.plot.markers.vertical_line(x=30)
>>> s.add_marker(m, permanent=True)
```

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)
```
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

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.

- **_do_snap_size**(value=None)
- **_get_patch_xy**()
  Returns the xy coordinates of the patch. In this implementation, the patch is centered on the position.
- **_get_resizer_pos**()
  Get the positions of the resizer handles.
- **_onmousemove**(event)
  On mouse motion move the patch if picked
- **_set_axes**(axes)
- **_set_patch**()
  Sets the patch to a matplotlib Circle with the correct geometry. The geometry is defined by _get_patch_xy, and size.
- **_set_size**(value)
  Setter for the ‘size’ property. Calls _size_changed to handle size change, if the value has changed.
- **_update_patch_geometry**()
  Updates all geometry of the patch on the plot.
- **_update_patch_position**()
  Updates the position of the patch on the plot.
- **_update_patch_size**()
  Updates the size of the patch on the plot.
- **_validate_pos**(value)
  Constrict the position within bounds.
- **decrease_size**()
  Decrement all sizes by one step. 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_size_in_indices**()
  Gets the size property converted to the index space (via ‘axes’ attribute).
- **increase_size**()
  Increment all sizes by one step. Applied via ‘size’ property.

**hyperspy.drawing._widgets.horizontal_line module**

**class hyperspy.drawing._widgets.horizontal_line.HorizontalLineWidget**(axes_manager, **kwargs)

**Bases:** hyperspy.drawing.widget.Widget1DBase

A draggable, horizontal line widget.

- **_onmousemove**(event)
  On mouse motion draw the cursor if picked
- **_set_patch**()
  Create the matplotlib patch(es), and store it in self.patch

3.1. hyperspy package
_update_patch_position()
   Updates the position of the patch on the plot.

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.

_get_string()
_onmousemove(event)
   on mouse motion draw the cursor if picked

_set_axes(axes)

_set_patch()
   Create the matplotlib patch(es), and store it in self.patch

_set_position(position)
   Sets the position of the widget (by values). The dimensions should correspond to that of the `axes` attribute.
   Calls _pos_changed if the value has changed, which is then responsible for triggering any relevant events.

_set_string(value)

_update_patch_position()
   Updates the position of the patch on the plot.

_update_patch_string()

_validate_pos(pos)
   Validates the passed position. Depending on the position and the implementation, this can either fire a ValueError, or return a modified position that has valid values. Or simply return the unmodified position if everything is ok.

   This default implementation bounds the position within the axes limits.

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).

Notes

This widget’s internal position does not lock to axes points by default.

Notes

The ‘position’ is now a 2D tuple: tuple((x1, x2), (y1, y2))

Notes

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
```

```python
_do_snap_position(value=None)
    Snaps position to axes grid. Returns snapped value. If value is passed as an argument, the internal state is left untouched, if not the position attribute is updated to the snapped value.

_do_snap_size(value=None)

_get_diff(event)
    Get difference in position in event and what is stored in _prev_pos, in value space.

_get_func_from_pos(cx, cy)
    Get interaction function from pixel position (cx, cy)

_get_line_normal()

_get_vertex(event)
    Check bitfield on self.func, and return vertex index.

_get_width_indicator_coords()
    Get coordinates of width indicators.

    The returned format is: [[[x0A, y0A], [x1A, y1A]], [[x0B, y0B], [x1B, y1B]]] Where A and B refer to the two lines

_move(event)
    Move line by drag start position + difference in mouse post from when dragging started (picked).

_onmousemove(event)
    Delegate to _move(), _resize() or _rotate().

_remove_size_patch()
_resize event
Move vertex by difference from pick / last mouse move. Update ‘_prev_pos’.

_rotate event
Rotate original points by the angle between mouse position and rotation start position (rotation center = line center).

_set_axes axes

_set_patch
Creates the line, and also creates the width indicators if appropriate.

_set_size value
Setter for the ‘size’ property.

_calls_size_changed to handle size change, if the value has changed.

_set_size_patch

_set_snap_size value

_update_patch_geometry
Set line position, and set width indicator’s if appropriate

_update_patch_position
Updates the position of the patch on the plot.

_update_patch_size
Updates the size of the patch on the plot.

_validate_pos pos
Make sure all vertices are within axis bounds.

_width_resize event

_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 mouse 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 the vectors ‘v1’ and ‘v2’.
Examples

```python
>>> 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**

```python
class hyperspy.drawing._widgets.range.ModifiableSpanSelector(ax, **kwargs)

    Bases: matplotlib.widgets.SpanSelector

    _get_mouse_position(event)

    _get_point_size_in_data_units()

    _get_range()

    _get_span_width()

    _get_span_x()

    _set_range(value)

    _set_span_width(value)

    _set_span_x(value)

    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()
```

3.1. hyperspy package
**update** (*args*)

Draw using blit() or draw_idle(), depending on self.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.

_add_patch_to (ax)

Create and add the matplotlib patches to ‘ax’

_get_range ()

_parse_bounds_args (args, kwargs)

_set_snap_position (value)

_set_snap_size (value)

_span_changed (widget)

_update_patch_geometry ()

Updates all geometry of the patch on the plot.

_update_patch_position ()

Updates the position of the patch on the plot.

_update_patch_size ()

Updates the size of the patch on the plot.

_validate_geometry (xl=None)

Make sure the entire patch always stays within bounds. First the position (either from position property or from xl argument), is limited within the bounds. Then, if the right edge are out of bounds, the position is changed so that they will be at the limit.

The modified geometry is stored, but no change checks are performed. Call _apply_changes after this in order to process any changes (the size might change if it is set larger than the bounds size).

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

```python
class hyperspy.drawing._widgets.rectangles.RectangleWidget(axes_manager, **kwargs)
```

`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.

```python
_decrease_xsize()
_decrease_ysize()
_get_patch_xy()
_increase_xsize()
_increase_ysize()
_onmousemove(event)
_parse_bounds_args(args, kwargs)
_set_a_size(idx, value)
_set_size(value)
```

Setter for the ‘size’ property.

Calls `_size_changed` to handle size change, if the value has changed.
_update_patch_geometry()
Updates all geometry of the patch on the plot.

_update_patch_position()
Updates the position of the patch on the plot.

_validate_geometry(x1=None, y1=None)
Make sure the entire patch always stays within bounds. First the position (either from position property or
from x1/y1 arguments), is limited within the bounds. Then, if the bottom/right edges are out of bounds,
the position is changed so that they will be at the limit.

The modified geometry is stored, but no change checks are performed. Call _apply_changes after this in
order to process any changes (the size might change if it is set larger than the bounds size).

_validate_pos(value)
Constrict the position 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:
• ‘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 vi-
sual, 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 Documentation, Release 1.6.0

_onmousemove_(event)
on mouse motion move the patch if picked

_set_patch()
Sets the patch to a matplotlib Rectangle with the correct geometry. The geometry is defined by _get_patch_xy, and get_size_in_axes.

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.

Parameters

• ax (matplotlib axes) – The axes where to draw the scale bar.
• units (str) –
• 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) –
• (None, (float, float)) (position) – 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.

_set_position(x, y)
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.

_onmousemove(event)
   on mouse motion draw the cursor if picked

_set_patch()
   Create the matplotlib patch(es), and store it in self.patch

_update_patch_position()
   Updates the position of the patch on the plot.

Module contents

Submodules

hyperspy.drawing.figure module

class hyperspy.drawing.figure..BlittedFigure

Bases: object

_draw_animated()
   Draw animated plot elements

_on_blit_draw(*args)

_on_close()

_update_animated()

add_marker(marker)

close()

create_figure(**kwargs)
   Create matplotlib figure

   Parameters **kwargs – All keyword arguments are passed to plt.figure.

remove_markers(render_figure=False)
   Remove all markers

property title
class hyperspy.drawing.image.ImagePlot(title='')
    Bases: hyperspy.drawing.figure.BlittedFigure
    _add_colorbar()
    _calculate_aspect()
    _calculate_vmin_max(data, auto_contrast=False, vmin=None, vmax=None)
    _update()
    _update_data()
    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 matplotlib.pyplot.figure().
    disconnect()
    gui_adjust_contrast(display=True, toolkit=None)
        Display widgets to adjust image contrast 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.
    on_key_press(event)
    optimize_colorbar(number_of_ticks=5, tolerance=5, step_prec_max=1)
    plot(data_function_kwargs={}, **kwargs)
    property scalebar
    set_contrast(vmin, vmax)
    set_quantity_label()
    toggle_norm()
    update(data_changed=True, auto_contrast=None, vmin=None, vmax=None, **kwargs)
Parameters

- **data_changed**(bool, optional) – Fetch and update the data to display. It can be used to avoid unnecessarily reading of the data from disk with working with lazy signal. The default is True.

- **auto_contrast**(bool or None, optional) – Force automatic resetting of the intensity limits. If None, the intensity values will change when `v` is in autoscale. Default is None.

- **vmax**(vmin,) – vmin and vmax are used to normalise the displayed data.

- ****kwargs**(dict) – The kwargs are passed to matplotlib.pyplot.imshow().

Raises **ValueError** – When the selected norm is not valid or the data are not compatible with the selected norm.

```python
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

```python
_get_data_shape()
_is_marker_static()
_render_figure()
_to_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.

```python
close(render_figure=True)
```

Remove and disconnect the marker.

**Parameters render_figure**(bool, optional, default True) – If True, the figure is rendered after removing the marker. If False, the figure is not rendered after removing the marker. This is useful when many markers are removed from a figure, since rendering the figure after removing each marker will slow things down.

```python
get_data_position(ind)
isiterable(obj)
```

**property marker_properties**

```python
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.

- **dict2marker** *(marker_dict, marker_name)*
  hyperspy.drawing.marker.dict2marker

- **markers_metadata_dict_to_markers** *(metadata_markers_dict, axes_manager)*
  hyperspy.drawing.marker.markers_metadata_dict_to_markers

**hyperspy.drawing.mpl_he module**

class hyperspy.drawing.mpl_he.MPL_HyperExplorer
  Bases: object

  _get_navigation_sliders()
  _on_navigator_plot_closing()
  _on_signal_plot_closing()
  assign_pointer()

  close()
  When closing, we make sure: - close the matplotlib figure - drawing events are disconnected - the attribute 'signal_plot' and 'navigation_plot' are set to None

  close_navigator_plot()

  property is_active
  A plot is active when it has the figure open meaning that it has either one of 'signal_plot' or 'navigation_plot' is not None and it has a attribute 'figure' which is not None.

  plot(**kwargs)
  plot_navigator(*title=None, **kwargs)*

  Parameters

  - **title** *(str, optional)* – Title of the navigator. The default is None.
  - ****kwargs (dict) – The kwargs are passed to plot method of hyperspy.drawing.image.ImagePlot() or hyperspy.drawing.signal1d.Signal1DLine().

  plot_signal()
hyperspy.drawing.mpl_hie module

class hyperspy.drawing.mpl_hie.MPL_HyperImage_Explorer
    Bases: hyperspy.drawing.mpl_he.MPL_HyperExplorer
    plot_signal(**kwargs)
        Parameters **kwargs (dict) – The kwargs are passed to plot method of the image figure.

class hyperspy.drawing.mpl_hse module

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.

class hyperspy.drawing.signal module

class hyperspy.drawing.signal.Signal1DFigure (title '')
    Bases: hyperspy.drawing.figure.BlittedFigure
    _on_close ()
    add_line (line, ax='left')
    create_axis ()
    create_right_axis ()
    plot (data_function_kwars= {}, **kwargs)
class hyperspy.drawing.signal1d.Signal1DLine

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.

_auto_update_line(*args, **kwargs)

Updates the line plot only if auto_update is True.

This is useful to connect to events that automatically update the line.

_get_data(real_part=False)

close()

property color

property get_complex

property line_properties

plot(data=1, **kwargs)

set_line_properties(**kwargs)

property type

update(force_replot=False, render_figure=True)

Update the current spectrum figure

hyperspy.drawing.signal1d._plot_component(factors, idx, ax=None, cal_axis=None, comp_label='PC')

hyperspy.drawing.signal1d._plot_loading(loadings, idx, axes_manager, ax=None, comp_label='PC', no_nans=True, calibrate=True, cmap=<matplotlib.colors.LinearSegmentedColormap object>)
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)

hyperspy.drawing.utils module

class hyperspy.drawing.utils.ColorCycle
    Bases: object
    _color_cycle = [(0.0, 0.0, 1.0, 1), (0.0, 0.5, 0.0, 1), (1.0, 0.0, 0.0, 1), (0.0, 0.75, 0.75, 1), (0.75, 0.0, 0.75, 1), (0.75, 0.75, 0.0, 1), (0.0, 0.0, 0.0, 1)]

hyperspy.drawing.utils._make_cascade_subplot(spectra, ax, color='blue', line_style='-', padding=1)

hyperspy.drawing.utils._make_heatmap_subplot(spectra, **plot_kwargs)

hyperspy.drawing.utils._make_overlap_plot(spectra, ax, color='blue', line_style='--')

hyperspy.drawing.utils._plot_spectrum(spectrum, ax, color='blue', line_style='--')

hyperspy.drawing.utils._set_spectrum_xlabel(spectrum, ax)

hyperspy.drawing.utils._transpose_if_required(signal, expected_dimension)

hyperspy.drawing.utils.animate_legend(fig=None, ax=None)
    Animate the legend of a figure.
    A spectrum can be toggled on and off by clicking on the line in the legend.
    Parameters
    - **fig**(None, matplotlib.figure, optional) – If None pick the current figure using “plt.gcf”.
    - **ax**(None, matplotlib.axes, optional) – 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 \( \textbf{vmax} (\textbf{vmin}) \) – The range of data to display.
    Returns \( \textbf{cvmin}, \textbf{cvmax} \) – The values to obtain a centre colormap.
    Return type scalar

hyperspy.drawing.utils.contrast_stretching(data, vmin=None, vmax=None)
    Estimate bounds of the data to display.
    Parameters
    - **data**(numpy array) –
• **vmax** *(vmin,)* – If str, formatted as ‘xth’, use this value to calculate the percentage of pixels that are left out of the lower and upper bounds. For example, for a vmin of ‘1th’, 1% of the lowest will be ignored to estimate the minimum value. Similarly, for a vmax value of ‘1th’, 1% of the highest value will be ignored in the estimation of the maximum value. See `numpy.percentile()` for more explanation. If None, use the percentiles value set in the preferences. If float of integer, keep this value as bounds.

**Returns** `vmin`, `vmax` – The low and high bounds.

**Return type** scalar

**Raises** `ValueError` – if the value of `vmin` `vmax` is out of the valid range for percentile calculation (in case of string values).

```python
def 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 *((None, string), optional)* –
    * _on_figure_window_close *((None, function), optional)* –
    * disable_xyscale_keys *(bool, optional)* – Disable the k, l and L shortcuts which toggle the x or y axis between linear and log scale. Default False.

    **Returns** `fig`

    **Return type** `plt.figure`
```

```python
def key_press_handler_custom(event, canvas):
    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, iterable), optional)* – 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 *(bool, optional)* – True if RGB colors are given in 8-bit [0 to 255] or False if given in arithmetic basis [0 to 1] (default).
    * register *(bool, optional)* – Wwitch to control whether or not to register the custom colormap with matplotlib in order to enable use by just the name string.

```python
def 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.picker_kwargs(value, kwargs= {})
  
 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'], optional) –

• `dont_plot` (bool, optional) – Default False.

Returns array

Return type RGB matrix

 hyperspy.drawing.utils.plot_histograms(signal_list, bins='fd', range_bins=None, color=None, line_style=None, legend='auto', fig=None, **kwargs)

Plot the histogram of every signal in the list in one figure.

This function creates a histogram for each signal and plots the list with the `utils.plot.plot_spectra` function.

Parameters

• `signal_list` (iterable) – Ordered list of spectra to plot. If style is “cascade” or “mosaic”, the spectra can have different size and axes.

• `bins` ({int, list, str}, optional) – If bins is a string, then it must be one of: ‘knuth’ : use Knuth’s rule to determine bins, ‘scott’ : use Scott’s rule to determine bins, ‘fd’ : use the Freedman-diaconis rule to determine bins, ‘blocks’ : use bayesian blocks for dynamic bin widths.

• `range_bins` ((None, tuple), optional) – The minimum and maximum range for the histogram. If not specified, it will be (x.min(), x.max()).

• `color` ((None, valid matplotlib color, list of colors), optional) – Sets the color of the lines of the plots. For a list, if its length is less than the number of spectra to plot, the colors will be cycled. If `None`, use default matplotlib color cycle.

• `line_style` ((None, valid matplotlib line style, list of line styles),) –

• `optional` – The main line styles are ‘-‘,’--‘,’steps‘,’-‘,’:‘. For 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, list of str, '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 toggled on and off by clicking on the line in the legend.

• `fig` ((None, matplotlib figure), optional) – If None, a default figure will be created.

• **kwargs – other keyword arguments (weight and density) are described in `numpy.histogram()`.

394 Chapter 3. hyperspy
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, suptitle=None, suptitle_fontsize=18, colorbar='multi', centre_colormap='auto', scalebar=None, scalebar_color='white', axes_decor='all', padding=None, tight_layout=False, aspect='auto', min_asp=0.1, name_frac_thresh=0.4, fig=None, vmin=None, vmax=None, **kwargs)*

Plot multiple images as sub-images in one figure.

Parameters:

- **images** *(list of Signal2D or BaseSignal)* – images should be a list of Signals to plot. For BaseSignal with navigation dimensions 2 and signal dimension 0, the signal will be transposed to form a Signal2D. Multi-dimensional images will have each plane plotted as a separate image. If any of the signal shapes is not suitable, a ValueError will be raised.
- **cmap** *(matplotlib colormap, list, 'mpl_colors'*, optional) – The colormap used for the images, by default uses the setting color map signal from the plot preferences. A list of colormaps can also be provided, and the images will cycle through them. Optionally, the value 'mpl_colors' will cause the cmap to loop through the default matplotlib colors (to match with the default output of the plot_spectra() method). Note: if using more than one colormap, using the 'single' option for colorbar is disallowed.
- **no_nans** *(bool, optional) – If True, set nans to zero for plotting.
- **per_row** *(int, optional)* – The number of plots in each row.
- **label** *(None, str, list of str*, optional) – Control the title labeling of the plotted images. If None, no titles will be shown. If ‘auto’ (default), function will try to determine suitable titles using Signal2D titles, falling back to the ‘titles’ option if no good short titles are detected. Works best if all images to be plotted have the same beginning to their titles. If ‘titles’, the title from each image’s metadata.General.title will be used. If any other single str, images will be labeled in sequence using that str as a prefix. If a list of str, the list elements will be used to determine the labels (repeated, if necessary).
- **labelwrap** *(int, optional)* – Integer specifying the number of characters that will be used on one line. If the function returns an unexpected blank figure, lower this value to reduce overlap of the labels between figures.
- **suptitle** *(str, optional) – Title to use at the top of the figure. If called with label='auto', this parameter will override the automatically determined title.
- **suptitle_fontsize** *(int, optional)* – Font size to use for super title at top of figure.
- **colorbar** *(‘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 particularly useful when using diverging color schemes. If “auto” (default), diverging color schemes are automatically centred.

- **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, dict, optional) – This parameter controls the spacing between images. If None, default options will be used. Otherwise, supply a dictionary with the spacing options as keywords and desired values as values. Values should be supplied as used in matplotlib.pyplot.subplots_adjust(), and can be ‘left’, ‘bottom’, ‘right’, ‘top’, ‘wspace’ (width) and ‘hspace’ (height).

- **tight_layout** (bool, optional) – If true, hyperspy will attempt to improve image placement in figure using matplotlib’s tight_layout. If false, repositioning images inside the figure will be left as an exercise for the user.

- **aspect** (str, 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 str, formatted as ‘xth’, use this value to calculate the percentage of pixels that are left out of the lower and upper bounds. For example, for a vmin of ‘1th’, 1% of the lowest will be ignored to estimate the minimum value. Similarly, for a vmax value of ‘1th’, 1% of the highest value will be ignored in the estimation of the maximum value. It must be in the range [0, 100] See numpy.percentile() for more explanation. If None, use the percentiles value set in the preferences. If float of integer, keep this value as bounds.

- **optional** (**kwargs,**) – Additional keyword 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`.

```python
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, False), default "True", optional) – 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` ("auto", None, "spectrum", "slider", BaseSignal), optional,)–
  - "auto" (default) – See signal.plot docstring for full description.

- `navigator_list` ({None, list of navigator arguments}, optional, default None) – Set different navigator options for the signals. Must use valid navigator arguments: “auto”, None, “spectrum”, “slider”, or a HyperSpy Signal. The list must have the same size as signal_list. If None, the argument specified in navigator will be used.

- `**kwargs` – 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, sll])
```

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")
```
```python
>>> 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, auto_update=None, **kwargs)`

Plot several spectra in the same figure.

**Parameters**

- **spectra** *(list of Signal1D or BaseSignal)* – Ordered spectra list of signal to plot. If `style` is “cascade” or “mosaic”, the spectra can have different size and axes. For `BaseSignal` with navigation dimensions 1 and signal dimension 0, the signal will be tranposed to form a `Signal1D`.

- **style** *({'overlap', 'cascade', 'mosaic', 'heatmap'})* – The style of the plot.

- **color** *(None, matplotlib color, list of colors, optional)* – Sets the color of the lines of the plots (no action on ‘heatmap’). For a list, if its length is less than the number of spectra to plot, the colors will be cycled. If `None`, use default matplotlib color cycle.

- **line_style** *(None, matplotlib line style, list of line_styles, optional)* – Sets the line style of the plots (no action on ‘heatmap’). The main line style are ‘-’, ‘–’, ‘steps’, ‘-.’, ‘:’. For 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’, ‘-.’, ‘:’.

- **padding** *(float, optional, default 1.0)* – Option for “cascade”. 1 guarantees that there is no overlapping. However, in many cases, a value between 0 and 1 can produce a tighter plot without overlapping. Negative values have the same effect but reverse the order of the spectra without reversing the order of the colors.

- **legend** *(None, list of str, 'auto'), optional)* – 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, optional)* – If True (default), a spectrum can be toggled on and off by clicking on the legended line.

- **legend_loc** *(str, int), optional)* – This parameter controls where the legend is placed on the figure; see the `pyplot.legend` docstring for valid values

- **fig** *(None, matplotlib figure, optional)* – If None, a default figure will be created. Specifying fig will not work for the ‘heatmap’ style.

- **ax** *(None, matplotlib ax (subplot), optional)* – If None, a default ax will be created. Will not work for ‘mosaic’ or ‘heatmap’ style.

- **auto_update** *(bool or None)* – If True, the plot will update when the data are changed. Only supported with style=’overlap’ and a list of signal with navigation dimension 0. If None (default), update the plot only for style=’overlap’.

- **optional (**)kwargs** – Keywords arguments passed to matplotlib.pyplot.figure() or matplotlib.pyplot.subplots() if style=’mosaic’. 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`: The matplotlib axis to change.
- `hs_axis`: The data axis that contains the values which 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

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)

_do_snap_position(value=None)

Snaps position to axes grid. Returns snapped value. If value is passed as an argument, the internal state is left untouched, if not the position attribute is updated to the snapped value.

_get_indices()

Returns a tuple with the position (indices).

_get_position()

Provides the position of the widget (by values) in a tuple.

_on_navigate(axes_manager)

Callback for axes_manager’s change notification.
Callback for mouse movement. For dragging, the implementor would normally check that the widget is
picked, and that the event.inaxes Axes equals self.ax.

Callback when the position of the widget has changed. It triggers the relevant events, and updates the patch
position.

Sets the position of the widget (by indices). The dimensions should correspond to that of the ‘axes’ at-
ttribute. Calls _pos_changed if the value has changed, which is then responsible for triggering any relevant
events.

Sets the position of the widget (by values). The dimensions should correspond to that of the ‘axes’ attribute.
Calls _pos_changed if the value has changed, which is then responsible for triggering any relevant events.

Updates all geometry of the patch on the plot.

Updates the position of the patch on the plot.

Validates the passed position. Depending on the position and the implementation, this can either fire a
ValueError, or return a modified position that has valid values. Or simply return the unmodified position if
everything is ok.

This default implementation bounds the position within the axes limits.

whenever a mouse button is released

Connect to the matplotlib Axes’ events.

property indices
property position
property snap_position

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) _up-
date_patch_geometry(self) _set_patch(self)
_apply_changes(old_size, old_position)
   Evaluates whether the widget has been moved/resized, and triggers the correct events and updates the patch geometry. This function has the advantage that the geometry is updated only once, preventing flickering and the ‘changed’ event only fires once.

_do_snap_size(value=None)

_get_size()
   Getter for ‘size’ property. Returns the size as a tuple (to prevent unintended in-place changes).

_set_axes(axes)

_set_size(value)
   Setter for the ‘size’ property.
   Calls _size_changed to handle size change, if the value has changed.

_set_snap_all(value)

_set_snap_size(value)

_size_changed()
   Triggers resize and changed events, and updates the patch.

_update_patch_geometry()
   Updates all geometry of the patch on the plot.

_update_patch_size()
   Updates the size of the patch on the plot.

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)

onpick(event)

set_size_in_indices(value)
   Sets the size property converted to the index space (via ‘axes’ attribute).

property size

property snap_all

3.1. hyperspy package
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
        Property that determines whether the resizer handles should be used
        
        Type bool

    resize_color
        The color of the resize handles.
        
        Type matplotlib color

    resize_pixel_size
        Size of the resize handles in screen pixels. If None, it is set equal to the size of one ‘data-pixel’ (image
        pixel size).
        
        Type tuple or None

    resizer_picked
        Inidcates 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.
        
        Type False or int

    _add_patch_to(ax)
        Same as widget base, but also adds resizers if ‘resizers’ property is True.

    _get_resizer_offset()
        Utility for getting the distance from the boundary box to the center of the resize handles.

    _get_resizer_pos()
        Get the positions of the resizer handles.

    _get_resizer_size()
        Gets the size of the resizer handles in axes coordinates. If ‘resize_pixel_size’ is None, a size of one pixel
        will be used.

    _set_patch()
        Creates the resizer handles, irregardless of whether they will be used or not.

    _set_resizers(value, ax)
        Turns the resizers on/off, in much the same way that _set_patch works.

    _update_resizers()
        Update resizer handles’ patch geometry.

    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 when 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 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.

**_set_position** *(position)*

Sets the position of the widget (by values). The dimensions should correspond to that of the `axes` attribute. Calls _pos_changed if the value has changed, which is then responsible for triggering any relevant events.

**_validate_pos** *(pos)*

Validates the passed position. Depending on the position and the implementation, this can either fire a ValueError, or return a modified position that has valid values. Or simply return the unmodified position if everything is ok.

This default implementation bounds the position within the axes limits.

**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 signal_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.

**_get_patch_bounds** ()

Returns the bounds of the patch in the form of a tuple in the order left, top, width, height. In matplotlib, ‘bottom’ is used instead of ‘top’ as the naming assumes an upwards pointing y-axis, meaning the lowest value corresponds to bottom. However, our widgets will normally only go on images (which has an inverted y-axis in MPL by default), so we define the lowest value to be termed ‘top’.

**_get_patch_xy** ()

Returns the xy position of the widget. In this default implementation, the widget is centered on the position.

**_update_patch_geometry** ()

Updates all geometry of the patch on the plot.

**_update_patch_position** ()

Updates the position of the patch on the plot.

**_update_patch_size** ()

Updates the size of the patch on the plot.

**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.

- `_add_patch_to(ax)`
  - Create and add the matplotlib patches to ‘ax’

- `_get_axes()`
  - Wrapped version of DataAxis.index2value, which bounds the value inbetween axis.low_value and axis.high_value+axis.scale, and does not raise a ValueError.

- `_on_navigate(axes_manager)`
  - Callback for axes_manager’s change notification.

- `_set_axes(axes)`

- `_set_patch()`
  - Create the matplotlib patch(es), and store it in self.patch

- `_v2i(axis, v)`
  - Wrapped version of DataAxis.value2index, which bounds the index inbetween axis.low_index and axis.high_index+1, and does not raise a ValueError.

**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.io_plugins package**

**Submodules**

**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 spectra, (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 \( (index=\text{None}, \quad downsample=1, \quad \text{cutoff}_\text{at}_K\text{V}=\text{None}, \quad \text{lazy}=\text{False}) \)

Unpack the Delphi/Bruker binary spectral map and return numpy array in memory efficient way.

Pure python/numpy implementation – slow, or cython/memoryview/numpy implementation if compiled and present (fast) is used.

Parameters  

- \text{index} (\text{None or int}) – The index of hypermap in bcf if there is more than one hyper map in file.
- \text{downsample} (\text{int}) – Downsampling factor. Differently than block_reduce from skimimage.measure, the parser populates reduced array by suming results of pixels, thus having lower memory requiriments. Default is 1.
- \text{cutoff}_\text{at}_K\text{V} (\text{None or float}) – Value in keV to truncate the array at. Helps reducing size of array. Default is None.
- \text{lazy} (\text{bool}) – It True, returns dask.array otherwise a numpy.array. Default is False.

Returns result – Bruker hypermap, with \((y,x,E)\) shape.

Return type numpy.ndarray or dask.array.array

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.

Parameters spectrum (etree xml object) – etree xml object, where spectrum.attrib[‘Type’] should be ‘TRTSpectrum’.

energy_to_channel (energy, \text{kV}=\text{True})
convert energy to channel index, optional kwarg ‘\text{kV}’ (default: True) should be set to False if given energy units is in \text{V}

class hyperspy.io_plugins.bruker.HyperHeader(xml_str, indexes, instrument=\text{None})
Bases: object

Wrap Bruker HyperMaping xml header into python object.

Arguments: \text{xml\_str} – the uncompressed to be provided with extracted Header xml from bcf. \text{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 through image index.

_parse_image (xml_node, overview=False)
parse image from bruker xml image node.

_set_elements (root)
wrap objectified xml part with selection of elements to self.elements list

_set_images (root)
Wrap objectified xml part with image to class attributes for self.image.
HyperSpy Documentation, Release 1.6.0

_set_microscope(root)
set microscope metadata from objectified xml part (TRTSEMDa, TRTSEMStageData, TRTDSConfiguration).

BCF can contain basic parameters of SEM column, and optionally the stage. This metadata can be not fully or at all available to Esprit and saved into bcf file as it depends on license and the link and implementation state between the microscope’s software and Bruker system.

_set_mode(instrument=None)
_set_sum_edx(root, indexes)
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.

Parameters
index (int) – Index of the map if multiply hypermaps are present in the same bcf.

Returns
Return type 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.

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.

Parameters
• index (int) – Index of the hypermap if multiply hypermaps are present in the same bcf. (default 0)
• downsample (int) – Downsample factor. (default 1)
• for_numpy (bool) – If False produce unsigned, otherwise signed 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 depth – numpy dtype large enough to use in final hypermap numpy array.

Return type numpy.dtype

gen_hspy_item_dict_basic()

get_acq_instrument_dict(detector=False, **kwargs)
return python dictionary with acquisition 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

3.1. hyperspy package
Methods: read_piece, setup_compression_metadata, get_iter_and_properties, get_as_BytesIO_string

_**calc_pointer_table_size**()

_**filetime_to_unix** (time)

Return recalculated windows filetime to unix time.

_**fill_pointer_table**()

Parse the sfs and populate self.pointers table.

self.pointer is the sfs pointer table containing addresses of every chunk of the file.

The pointer table if the file is big can extend through many sfs chunks. Differently than files, the pointer table of file have no table of pointers to the chunks. Instead if pointer table is larger than sfs chunk, the chunk header contains next chunk number (address can be calculated using known chunk size and global offset) with continuation of file pointer table, thus it have to be read and filled consecutive.

_**iter_read_chunks** (first=0)

Generate and return iterator for reading and returning sfs internal file in chunks.

By default it creates iterator for whole file, however with kwargs ‘first’ and ‘chunks’ the range of chunks for iterator can be set.

Keyword arguments: first – the index of first chunk from which to read. (default 0) chunks – the number of chunks to read. (default False)

_**iter_read_compr_chunks**()

Generate and return reader and decompressor iterator for compressed with zlib compression sfs internal file.

Returns: iterator of decompressed data chunks.

_**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.BytesIO 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.
filename

get_file()

_check_the_compresion(temp_item_list)
parse, check and setup the self.compression

_flat_items_to_dict(paths, temp_item_list)
place items from flat list into dictionary tree of virtual file system

_setup_vfs()
Setup the virtual file system tree represented as python dictionary with values populated with SFSTreeItem instances

See also:
SFSTreeItem()

get_file(path)
Return the SFSTreeItem (aka internal file) object from sfs container.

Parameters

path(str) – 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.

Return type

SFSTreeItem

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()

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.

Parameters

• select_type(str or None) – 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(int, None or str) – 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** (*int*) – 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** (*int, float or None*) – 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 or None*) – Can be either ‘TEM’ or ‘SEM’. Default is None.

```
hyperspy.io_plugins.bruker.dictionarize(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.

**Parameters**

- **virtual file handle returned by SFS_reader instance**
  *(virtual_file)* – or by object inheriting it (e.g. BCF_reader instance)

- **numpy shape** *(shape)* –

- **numpy dtype** *(dtype)* –

- **downsample factor** *(downsample)* –

  **note!** *(downsample, shape and dtype are interconnected and needs)* –

  **be properly calculated otherwise wrong output or segfault (to)* –

  **expected(is)* –

**Returns**

**Return type**  numpy array of bruker hypermap, with (y, x, E) shape.
`hyperspy.io_plugins.bruker`\._spx_reader\( \text{filename, lazy=False} \)

**hyperspy.io_plugins.dens module**

`hyperspy.io_plugins.dens._bad_file\( \text{filename} \)`

`hyperspy.io_plugins.dens._cnv_time\( \text{timestr} \)`

`hyperspy.io_plugins.dens.file_reader\( \text{filename, *args, **kwds} \)`

**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.
    dm_version, endian, tags_dict
    parse_file, parse_header, get_image_dictionaries
    _complex_type = (15, 18, 20)
    check_data_tag_delimiter()
    find_next_data_tag()
    find_next_tag()
    get_data_reader\( \text{enc_dtype} \)
    get_image_dictionaries()
    Returns the image dictionaries of all images in the file except the thumbnails.
    Returns
    Return type dict, None
    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.
    parse_file()
    parse_header()
    parse_string_definition()
    Reads and returns the length of the string.
    The position in the file must be just after the string encoded dtype.
    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\( \text{size=False} \)
    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()
```

3.1. hyperspy package
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

_get_EELS_exposure_time(tags)
_get_data_array()
_get_date(date)
_get_microscope_name(ImageTags)
_get_mode(mode)
_get_quantity(units)
_get_time(time)
_parse_string(tag)
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.__get_ipr_header(f, endianess)
Get the header of an spc file, checking for the file version as necessary

Parameters

- f (file) – A file object for the .spc file to be read (i.e. file should be already opened with open())
- endianess (char) – Byte-order of data to read

Returns ipr_header – Array containing the binary information read from the .ipr file

Return type np.ndarray

hyperspy.io_plugins.edax.__get_spc_header(f, endianess, load_all_spc)
Get the header of an spc file, checking for the file version as necessary

Parameters

- f (file) – A file object for the .spc file to be read (i.e. file should be already opened with open())
- 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 parts relevant to HyperSpy

Returns spc_header – Array containing the binary information read from the .spc file

Return type np.ndarray
hyperspy.io_plugins.edax._add_spc_metadata(metadata, spc_header)
Return metadata with information from the .spc header added

Parameters

- **metadata** *(dict)* – current metadata of signal without spectral calibration information added
- **spc_header** *(dict)* – header of .spc file that contains spectral information such as azimuth and elevation angles, energy resolution, etc.

Returns **metadata** – copy of original dictionary with spectral calibration added

Return type **dict**

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 (milliseec)*
- 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]*
- 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)
- endianess (char) – byte-order used to read the data
- version (float) – version of syc 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
- collectDateDay: 1 byte char; Day the spectrum was collected
- collectDateMon: 1 byte char; Month the spectrum was collected
- collectTimeMin: 1 byte char; Minute the spectrum was collected
- collectTimeHour: 1 byte char; Hour 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=Eucentric Cross, 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=clock, 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 each ROI*
- **filler5_2**: 1 byte;
HyperSpy Documentation, Release 1.6.0

- 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*
- bgPercent: 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:
HyperSpy Documentation, Release 1.6.0

- 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*

```python
dtype_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 SPD file header.

**Return type** *list*

```python
dtype_list = hyperspy.io_plugins.edax.get_spd_dtype_list(endianess='<')
```

**hyperspy.io_plugins.edax.spc_reader** *(filename, endianess='<', load_all_spc=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

**Returns** – Remaining arguments are passed to the Numpy memmap function

**Return type** *list*

```python
data = hyperspy.io_plugins.edax.spc_reader(filename, endianess='<', load_all_spc=False, **kwargs)
```

**hyperspy.io_plugins.edax.spd_reader** *(filename, endianess='<', spc_fname=None, ipr_fname=None, load_all_spc=False, **kwargs)*

Read data from an SPD spectral map specified by filename.

**Parameters**

- **filename** *(str)* – Name of SPD 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

**Returns** – Remaining arguments are passed to the Numpy memmap function

**Return type** *list*
Parameters

- **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 No, 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 No, 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

hyperspy.io_plugins.emd module

class hyperspy.io_plugins.emd.EMD(*signals=None, user=None, microscope=None, sample=None, comments=None*)

Bases: object

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://emdatasets.com/).

**signals**

Dictionary which contains all datasets as Signal instances.

**user**

Dictionary which contains user related metadata.

**microscope**

Dictionary which contains microscope related metadata.

**sample**

Dictionary which contains sample related metadata.

**comments**

Dictionary which contains additional commentary metadata.
**Type** dictionary

```python
_log = <Logger hyperspy.io_plugins.emd (WARNING)>
_read_signal_from_group(name, group, lazy=False)
_write_signal_to_group(signal_group, signal)
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!

```python
classmethod load_from_emd(filename, lazy=False, dataset_name=None)
```

Construct EMD object from an emd-file.

**Parameters**

- **filename** (*str*) – The name of the emd-file from which to load the signals. Standard file extension is `.emd`.
- **False** (*bool, optional*) – If False (default) loads data to memory. If True, enables loading only if requested.
- **dataset_name** (*str or iterable, optional*) – Only add dataset with specific name. Note, this has to be the full group path in the file. For example `/experimental/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

```python
log_info()
```

( all relevant information about the EMD instance.

```python
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.EMD_NCEM
    Bases: object

    Class for reading and writing the Berkeley variant of the electron microscopy datasets (EMD) file format. It reads files EMD NCEM, including files generated by the prismatic software.

    dictionaries
        List of dictionaries which are passed to the file_reader.

        Type: list

    static _get_emd_group_type(group)
        Return the value of the 'emd_group_type' attribute if it exist, otherwise returns False

    property _is_prismatic_file

    property _is_py4DSTEM_file

    _parse_attribute(obj, key)

    static _parse_axis(axis_data)
        Estimate, offset, scale from a 1D array

    _parse_metadata(group_basename, title='')

    _parse_original_metadata()

    _read_data_from_groups(group_path, dataset_name, stack_key=None, original_metadata={})

    _read_emd_version(group)
        Return the group version if the group is an EMD group, otherwise return None.

    _write_signal_to_group(signal_group, signal)

    classmethod find_dataset_paths(file)
        Find the paths of all groups containing valid EMD data.

        Returns: datasets – List of path to these group.

        Return type: list

    read_file(file, lazy=None, dataset_path=None, stack_group=None)
        Read the data from an emd file

        Parameters:
        • file (file handle) – Handle of the file to read the data from.
        • lazy (bool, optional) – Load the data lazily. The default is False.
        • dataset_path (None, str or list of str) – Path of the dataset. If None, load all supported datasets, otherwise the specified dataset. The default is None.
        • stack_group (bool, optional) – Stack datasets of groups with common name.

        Relevant for emd file version >= 0.5 where groups can be named 'group0000', 'group0001', etc.

    write_file(file, signal, **kwargs)
        Write signal to file.

        Parameters:
        • file (str of h5py file handle) – If str, filename of the file to write, otherwise a h5py file handle
• **signal** (*instance of hyperspy signal*) – The signal to save.
• **kwargs** (*dict*) – Dictionary containing metadata which will be written as attribute of the root group.

```python
class hyperspy.io_plugins.emd.FeiEMDReader(filename=None, 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

```python
_check_im_type()
_convert_datetime(unix_time)
_convert_element_list(d)
_convert_scale_units(value, units, factor=1)
_get_dispersion_offset(original_metadata)
_get_local_time_zone()
_get_mapping(map_selected_element=True, parse_individual_EDS_detector_metadata=True)
_get_metadata_dict(om)
_parse_frame_time(original_metadata, factor=1)
_parse_image_display()
_parse_metadata_group(group, group_name)
_read_data(select_type)
_read_image(image_group, image_sub_group_key)
    Return a dictionary ready to parse of return to io module
_read_images()
_read_single_spectrum()
_read_spectrum(spectrum_group, spectrum_sub_group_key)
_read_spectrum_stream()
read_file(f)
```

```python
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._get_detector_metadata_dict`(om, detector_name)

`hyperspy.io_plugins.emd._get_keys_from_group`(group)

`hyperspy.io_plugins.emd._parse_detector_name`(original_metadata)

`hyperspy.io_plugins.emd._parse_metadata`(data_group, sub_group_key)

`hyperspy.io_plugins.emd._parse_sub_data_group_metadata`(sub_data_group)

`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.

`hyperspy.io_plugins.emd.file_reader`(filename, lazy=False, **kwds)

Read EMD file, which can be a NCEM or a Velox variant of the EMD format.

**Parameters**

- `filename` (str) – Filename of the file to write.
- `lazy` (bool) – Open the data lazily. Default is False.
- `**kwds` (dict) – Keyword argument pass to the EMD NCEM or EMD Velox reader. See user guide or the docstring of the load function for more information.

`hyperspy.io_plugins.emd.file_writer`(filename, signal, **kwds)

Write signal to EMD NCEM file.

**Parameters**

- `file` (str of h5py file handle) – If str, filename of the file to write, otherwise a h5py file handle
- `signal` (instance of hyperspy signal) – The signal to save.
**kwargs (dict) – Dictionary containing metadata which will be written as attribute of
the root group.

`hyperspy.io_plugins.emd.is_EMD_NCEM(file)`

Parameters `file` (h5py file handle) – DESCRIPTION.

Returns DESCRIPTION.

Return type bool

`hyperspy.io_plugins.emd.is_EMD_Velox(file)`

Function to check if the EMD file is an Velox file.

Parameters `file` (string or HDF5 file handle) – The name of the emd-file from which
to load the signals. Standard file extension is ‘emd’.

Returns

Return type True if the file is a Velox file, otherwise False

`hyperspy.io_plugins.emd.read_emd_version(group)`

Function to read the emd file version from a group. The EMD version is saved in the attributes ‘version_major’
and ‘version_minor’.

Parameters `group` (hdf5 group) – The group to extract the version from.

Returns file version – Empty string if the file version is not defined in this group

Return type str

** hyperspy.io_plugins.empad module **

`hyperspy.io_plugins.empad._convert_scale_units(value, units, factor=1)`

`hyperspy.io_plugins.empad._parse_xml(filename)`

`hyperspy.io_plugins.empad._read_raw(info, fp, lazy=False)`

`hyperspy.io_plugins.empad.file_reader(filename, lazy=False, **kwds)`

** hyperspy.io_plugins.fei module **

`hyperspy.io_plugins.fei._get_date_time(value)`

`hyperspy.io_plugins.fei._get_microscope_name(value)`

`hyperspy.io_plugins.fei._get_simplified_mode(mode)`

`hyperspy.io_plugins.fei._guess_units_from_mode(objects_dict, header)`

`hyperspy.io_plugins.fei.convert_xml_to_dict(xml_object)`

`hyperspy.io_plugins.fei.dimension_array_dtype(n, DescriptionLength, UnitsLength)`

`hyperspy.io_plugins.fei.emi_reader(filename, dump_xml=False, **kwds)`

`hyperspy.io_plugins.fei.emixml2dtb(et, dictree)`

`hyperspy.io_plugins.fei.file_reader(filename, *args, **kwds)`

`hyperspy.io_plugins.fei.get_axes_from_position(header, data)`
**hyperspy.io_plugins.fei.**

*get_calibration_from_position*(position)

Compute the size, scale and offset of a linear axis from coordinates.

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**


**Returns**

axis_attr

**Return type**

dictionary with size, scale, offset keys.

**hyperspy.io_plugins.fei.**

got_data_dtype_list*(file, offset, record_by)*

got_data_tag_dtype_list*(data_type_id)*

got_header_dtype_list*(file)*

got_lengths*(file)*

got_xml_info_from_emi*(emi_file)*

guess_record_by*(record_by_id)*

**hyperspy.io_plugins.fei.**

load_only_data*(filename, array_shape, record_by, num_axes, data=None, header=None, only_valid_data=False)*

**hyperspy.io_plugins.fei.**

load_ser_file*(filename)*

**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.

```python
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

```python
hyperspy.io_plugins.image._read_data(filename)
hyperspy.io_plugins.image.file_reader(filename, **kwds)
```

Read data from any format supported by PIL.

**Parameters**
- `filename` *(str)*

```python
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

```python
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

```python
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 complain 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.nexus module**

Nexus file reading, writing and inspection.

**hyperspy.io_plugins.nexus._byte_to_string** *(value)*

Decode a byte string.

**Parameters**

• **value** *(byte str)* –

**Returns**

decoded version of input value

**Return type**

str

**hyperspy.io_plugins.nexus._check_search_keys** *(search_keys)*

**hyperspy.io_plugins.nexus._extract_hdf_dataset** *(group, dataset, lazy=False)*

Import data from hdf path.

**Parameters**

• **group** *(hdf group)* – group from which to load the dataset

• **dataset** *(str)* – path to the dataset within the group

• **lazy** *(bool (default: True))* – If true use lazy opening, if false read into memory

**Returns**

**Return type**

dask or numpy array

**hyperspy.io_plugins.nexus._find_data** *(group, search_keys=None, hardlinks_only=False)*

Read from a nexus or hdf file and return a list of the dataset entries.

The method iterates through group attributes and returns NXdata or hdf datasets of size >=2 if they’re not already NXdata blocks and returns a list of the entries. This is a convenience method to inspect a file to see which datasets are present rather than loading all the sets in the file as signals h5py.visit or visititems does not visit soft links or external links so an implementation of a recursive search is required. See https://github.com/h5py/h5py/issues/671

**Parameters**

• **group** *(hdf group or File)* –

• **search_keys** *(string, list of strings or None, default: None)* – Only return items which contain the strings. E.g search_list = [“instrument”, “Fe”] will return hdf entries with instrument or Fe in their hdf path.
• **hardlinks_only** *(bool, default: False)* — Option to ignore links (soft or external) within the file.

**Returns**
nx_dataset_list is a list of all NXdata paths
hdf_dataset_list is a list of all hdf_datasets not linked to an NXdata set.

**Return type**
nx_dataset_list, hdf_dataset_list

**hyperspy.io_plugins.nexus._find_search_keys_in_dict** *(tree, search_keys=None)*

Search through a dict for search keys.

This is a convenience method to inspect a file for a value rather than loading the file as a signal

**Parameters**

- **tree** *(h5py File object)* —
- **search_keys** *(string or list of strings)* — Only return items which contain the strings. e.g. search_keys = [“instrument”,“Fe”] will return hdf entries with instrument or Fe in their hdf path.

**Returns**
When search_list is specified only full paths containing one or more search_keys will be returned

**Return type**
dict

**hyperspy.io_plugins.nexus._fix_exclusion_keys** *(key)*

Exclude hyperspy specific keys.

Signal and DictionaryBrowser break if a key is a dict method - e.g. {“keys”:2.0}.

This method prepends the key with “fix_” so the information is still present to work around this issue

**key** : str

**Returns**

**Return type**
str

**hyperspy.io_plugins.nexus._getlink** *(h5group, rootkey, key)*

Return the link target path.

If a hdf group is a soft link or has a target attribute this method will return the target path. If no link is found return None.

**Returns**
Soft link path if it exists, otherwise None

**Return type**
str

**hyperspy.io_plugins.nexus._is_int** *(s)*

Check that s in an integer.

**s** : python object to test

**Returns**
True or False

**Return type**
bool

**hyperspy.io_plugins.nexus._is_linear_axis** *(data)*

Check if the data is linearly incrementing.

**data** : dask or numpy array

**Returns**
True or False

**Return type**
bool
hyperspy.io_plugins.nexus._is_numeric_data(data)
Check that data contains numeric data.

data : dask or numpy array

Returns True or False

Return type bool

hyperspy.io_plugins.nexus._load_metadata(group, lazy=False)
Search through a hdf group and return the group structure.

h5py.visit or visititems does not visit soft links or external links so an implementation of a recursive search is required. See https://github.com/h5py/h5py/issues/671

Parameters
• group (hdf group) – location to load the metadata from
• lazy (bool, default : False) – Option for lazy loading

Returns dictionary of group contents

Return type dict

hyperspy.io_plugins.nexus._nexus_dataset_to_signal(group, nexus_dataset_path, lazy=False)
Load an NXdata set as a hyperspy signal.

Parameters
• group (hdf group containing the NXdata) –
• nexus_data_path (str) – Path to the NXdata set in the group
• lazy (bool, default : True) – lazy loading of data

Returns A signal dictionary which can be used to instantiate a signal.

Return type dict

hyperspy.io_plugins.nexus._parse_from_file(value, lazy=False)
To convert values from the hdf file to compatible formats.

When reading string arrays we convert or keep string arrays as byte_strings (some io_plugins only supports byte-strings arrays so this ensures inter-compatibility across io_plugins) Arrays of length 1 - return the single value stored. Large datasets are returned as dask arrays if lazy=True.

Parameters
• value (input read from hdf file (array, list, tuple, string, int, float)) –
• lazy (bool {default: False}) – The lazy flag is only applied to values of size >=2

Returns parsed value.

Return type str, int, float, ndarray dask Array

hyperspy.io_plugins.nexus._parse_to_file(value)
Convert to a suitable format for writing to HDF5.

For example unicode values are not compatible with hdf5 so conversion to byte strings is required.

Parameters – input object to write to the hdf file

Returns
Return type  parsed value

`hyperspy.io_plugins.nexus._text_split(s, sep)`  
Split a string based on list of separators.

Parameters

- `s (str)` -  
- `sep (str - separator or list of separators e.g. '.') or ['_', '/'])` -  

Returns  String sections split based on the separators

Return type  list

`hyperspy.io_plugins.nexus._write_nexus_attr(dictionary, group)`  
Recursively iterate through dictionary and write “attrs” dictionaries.

This step is called after the groups and datasets have been created

Parameters

- `dictionary (dict)` – Input dictionary to be written to the hdf group
- `group (hdf group)` – location to store the attrs sections of the dictionary

`hyperspy.io_plugins.nexus._write_nexus_groups(dictionary, group, **kwds)`  
Recursively iterate through dictionary and write groups to nexus.

Parameters

- `dictionary (dict)` – dictionary contents to store to hdf group
- `group (hdf group)` – location to store dictionary
- `**kwds (additional keywords)` – additional keywords to pass to h5py.create_dataset method

`hyperspy.io_plugins.nexus._write_signal(signal, nxgroup, signal_name, **kwds)`  
Store the signal data as an NXdata dataset.

Parameters

- `signal (Hyperspy signal)` –  
- `nxgroup (HDF group)` – Entry at which to save signal data
- `signal_name (str)` – Name under which to store the signal entry in the file

`hyperspy.io_plugins.nexus.file_reader(filename, lazy=False, dataset_keys=None, metadata_keys=None, nxdata_only=False, hardlinks_only=False, use_default=False, **kwds)`  
Read NXdata class or hdf datasets from a file and return signal(s).

Note:  Loading all datasets can result in a large number of signals Please review your datasets and use the dataset_keys to target the datasets of interest. “keys” is a special keywords and prepended with “fix” in the metadata structure to avoid any issues.

Datasets are all arrays with size>2 (arrays, lists)
**dataset_keys** (None, str, list of strings, default : None) – If None all datasets are returned. If a string or list of strings is provided only items whose path contain the string(s) are returned. For example dataset_keys = ["instrument", “Fe”] will return data entries with instrument or Fe in their hdf path.

**metadata_keys** (: None, str, list of strings, default : None) – Only return items from the original metadata whose path contain the strings .e.g metadata_keys = ["instrument", “Fe”] will return all metadata entries with “instrument” or “Fe” in their hdf path.

**nxdata_only** (bool, default : False) – If True only NXdata will be converted into a signal if False NXdata and any hdf datasets will be loaded as signals

**hardlinks_only** (bool, default : False) – If True any links (soft or External) will be ignored when loading.

**use_default** (bool, default : False) – If True and a default NXdata is defined in the file load this as a signal. This will ignore the other keyword options. If True and no default is defined the file will be loaded according to the keyword options.

**Returns** dict

**Return type** signal dictionary or list of signal dictionaries

**See also:**

- list_datasets_in_file()
- read_metadata_from_file()

**hyperspy.io_plugins.nexus.file_writer** (filename, signals, save_original_metadata=True, use_default=False, *args, **kwds)

Write the signal and metadata as a nexus file.

This will save the signal in NXdata format in the file. As the form of the metadata can vary and is not validated it will be stored as an NXcollection (an unvalidated collection)

**Parameters**

- **filename**(str) – Path of the file to write
- **signals**(signal or list of signals) – Signal(s) to be written
- **save_original_metadata**(bool , default : False) – Option to save hyperspy.original_metadata with the signal. A loaded Nexus file may have a large amount of data when loaded which you may wish to omit on saving
- **use_default**(bool , default : False) – Option to define the default dataset in the file. If set to True the signal or first signal in the list of signals will be defined as the default (following Nexus v3 data rules).

**See also:**

- file_reader()
- list_datasets_in_file()
- read_metadata_from_file()

**hyperspy.io_plugins.nexus.list_datasets_in_file** (filename, dataset_keys=None, hardlinks_only=False, verbose=True)

Read from a nexus or hdf file and return a list of the dataset paths.
This method is used to inspect the contents of a Nexus file. The method iterates through group attributes and returns NXdata or hdf datasets of size >=2 if they’re not already NXdata blocks and returns a list of the entries. This is a convenience method to inspect a file to list datasets present rather than loading all the datasets in the file as signals.

**Parameters**

- `filename` *(str)* - path of the file to read
- `dataset_keys` *(str, list of strings or None, default: None)* - If a str or list of strings is provided only return items whose path contain the strings. For example, `dataset_keys = ["instrument", "Fe"]` will only return hdf entries with “instrument” or “Fe” somewhere in their hdf path.
- `hardlinks_only` *(bool, default : False)* - If true any links (soft or External) will be ignored when loading.
- `verbose` *(boolean, default : True)* - Prints the results to screen

**Returns** list of paths to datasets

**Return type** list

**See also:**

- `file_reader()`
- `file_writer()`
- `read_metadata_from_file()`

```python
hyperspy.io_plugins.nexus.read_metadata_from_file(filename, metadata_keys=None, lazy=False, verbose=False)
```

Read the metadata from a nexus or hdf file.

This method iterates through the file and returns a dictionary of the entries. This is a convenience method to inspect a file for a value rather than loading the file as a signal.

**Parameters**

- `filename` *(str)* - path of the file to read
- `metadata_keys` *(None, str or list_of_strings, default : None)* - None will return all datasets found including linked data. Providing a string or list of strings will only return items which contain the string(s). For example, `search_keys = ["instrument", "Fe"]` will return hdf entries with “instrument” or “Fe” in their hdf path.
- `verbose` *(bool, default : False)* - Pretty Print the results to screen

**Returns** Metadata dictionary.

**Return type** dict

**See also:**

- `file_reader()`
- `file_writer()`
- `list_datasets_in_file()`
class hyperspy.io_plugins.phenom.ElidReader(pathname, block_size=1048576)
    Bases: object
    
    _get_datetime(metadata)
    _get_local_time_zone()
    _get_unit(value)
    _get_unit_factor(unit)
    _get_value_with_unit(item)
    _make_image_dict(om, data, title)
    _make_line_spectrum_dict(om, offset, dispersion, data, title)
    _make_map_spectrum_dict(om, offset, dispersion, data, title)
    _make_mapping()
    _make_metadata_dict(signal_type, title, datetime)
    _make_spot_spectrum_dict(om, offset, dispersion, data, title)
    _read(size=1)
    _read_Analyses(label, metadata)
    _read_Analysis(label, am)
    _read_CommonAnalysis(am)
    _read_ConstructiveAnalyses()
    _read_ConstructiveAnalysis(label, metadata)
    _read_ConstructiveAnalysisSource()
    _read_ConstructiveAnalysisSources()
    _read_DifferenceAnalysis(label, am)
    _read_Images()
    _read_Image()
    _read_LineScanAnalysis(label, am)
    _read_MapAnalysis(label, am)
    _read_MsaAnalysis(label, am)
    _read_Project()
    _read_RegionAnalysis(label, am)
    _read_SpotAnalysis(label, am)
    _read_bool()
    _read_drift_correction()
    _read_eds_metadata(om)
    _read_element_families()
    _read_element_family()
_read_float64()
_write_float64s()
_write_int32()
_write_int32s()
_read_oxide()
_read_oxides()
_read_spectrum()
_read_string()
_read_tiff()
_read_uint32()
_read_uint8()
_read_uint8s()
_read_varuint32()

hyperspy.io_plugins.phenom.IsBZip2(pathname)

hyperspy.io_plugins.phenom.IsGZip(pathname)

hyperspy.io_plugins.phenom.element_symbol(z)

hyperspy.io_plugins.phenom.family_symbol(i)

hyperspy.io_plugins.phenom.file_reader(filename, log_info=False, lazy=False, **kwds)

**hyperspy.io_plugins.protochips module**

class hyperspy.io_plugins.protochips.ProtochipsCSV(filename)
    Bases: object
    _get_axes()
    _get_mapping()
    _get_metadata(quantity)
    _get_metadata_time_axis()
    _get_original_metadata()
    _is_protochips_csv_file()
    _parse_calibration_file_name(path)
    _parse_calibration_filepath()
    _parse_header()
    _parse_metadata_header(line)
    _parse_notes()
    _parse_quantity(quantity)
    _parse_quantity_units(quantity)
    _read_all_metadata_header(f)
_read_data()

get_dictionary(quantity)

hyperspy.ioplugins.protochips._protochips_log_reader(csv_file)

hyperspy.ioplugins.protochips.file_reader(filename, *args, **kwds)

**hyperspy.io_plugins.ripple module**

hyperspy.ioplugins.ripple.correct_INCA_format(fp)

hyperspy.ioplugins.ripple.file_reader(filename, rpl_info=None, encoding='latin-1', mmap_mode='c', *args, **kwds)

Parses a Lispix (http://www.nist.gov/lispx/) 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>
<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</td>
</tr>
<tr>
<td>height</td>
<td>int</td>
<td>number of rows</td>
</tr>
<tr>
<td>depth</td>
<td>int</td>
<td>number of images or spectral pts</td>
</tr>
<tr>
<td>offset</td>
<td>int</td>
<td>bytes to skip</td>
</tr>
<tr>
<td>data-type</td>
<td>str</td>
<td>‘signed’, ‘unsigned’, or ‘float’</td>
</tr>
<tr>
<td>data-length</td>
<td>str</td>
<td>bytes per pixel ‘1’, ‘2’, ‘4’, or ‘8’</td>
</tr>
<tr>
<td>byte-order</td>
<td>str</td>
<td>‘big-endian’, ‘little-endian’, or ‘dont-care’</td>
</tr>
<tr>
<td>record-by</td>
<td>str</td>
<td>‘image’, ‘vector’, or ‘dont-care’</td>
</tr>
<tr>
<td># X-ray keys:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ev-per-chan</td>
<td>int</td>
<td>optional, eV per channel</td>
</tr>
<tr>
<td>detector-peak-width-ev</td>
<td>int</td>
<td>optional, FWHM for the Mn K-alpha line</td>
</tr>
<tr>
<td># HyperSpy-specific keys</td>
<td></td>
<td></td>
</tr>
<tr>
<td>depth-origin</td>
<td>int</td>
<td>energy offset in pixels</td>
</tr>
<tr>
<td>depth-scale</td>
<td>float</td>
<td>energy scaling (units per pixel)</td>
</tr>
<tr>
<td>depth-units</td>
<td>str</td>
<td>energy units, usually eV</td>
</tr>
<tr>
<td>depth-name</td>
<td>str</td>
<td>Name of the magnitude stored as depth</td>
</tr>
<tr>
<td>width-origin</td>
<td>int</td>
<td>column offset in pixels</td>
</tr>
<tr>
<td>width-scale</td>
<td>float</td>
<td>column scaling (units per pixel)</td>
</tr>
</tbody>
</table>

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width-units</td>
<td>str</td>
<td>column units, usually nm</td>
</tr>
<tr>
<td>width-name</td>
<td>str</td>
<td>Name of the magnitude stored as width</td>
</tr>
<tr>
<td>height-origin</td>
<td>int</td>
<td>row offset in pixels</td>
</tr>
<tr>
<td>height-scale</td>
<td>float</td>
<td>row scaling (units per pixel)</td>
</tr>
<tr>
<td>height-units</td>
<td>str</td>
<td>row units, usually nm</td>
</tr>
<tr>
<td>height-name</td>
<td>str</td>
<td>Name of the magnitude stored as height</td>
</tr>
<tr>
<td>signal</td>
<td>str</td>
<td>Type of the signal stored, e.g. EDS_SEM</td>
</tr>
<tr>
<td>convergence-angle</td>
<td>float</td>
<td>TEM convergence angle in mrad</td>
</tr>
<tr>
<td>collection-angle</td>
<td>float</td>
<td>EELS spectrometer collection semi-angle in mrad</td>
</tr>
<tr>
<td>beam-energy</td>
<td>float</td>
<td>TEM beam energy in keV</td>
</tr>
<tr>
<td>elevation-angle</td>
<td>float</td>
<td>Elevation angle of the EDS detector</td>
</tr>
<tr>
<td>azimuth-angle</td>
<td>float</td>
<td>Elevation angle of the EDS detector</td>
</tr>
<tr>
<td>live-time</td>
<td>float</td>
<td>Live time per spectrum</td>
</tr>
<tr>
<td>energy-resolution</td>
<td>float</td>
<td>Resolution of the EDS (FHWM of MnKa)</td>
</tr>
<tr>
<td>tilt-stage</td>
<td>float</td>
<td>The tilt of the stage</td>
</tr>
<tr>
<td>date</td>
<td>str</td>
<td>date in ISO 8601</td>
</tr>
<tr>
<td>time</td>
<td>str</td>
<td>time in ISO 8601</td>
</tr>
<tr>
<td>title</td>
<td>str</td>
<td>title of the signal to be stored</td>
</tr>
</tbody>
</table>

**Note:** 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.

```python
hyperspy.io_plugins.ripple.file_writer(filename, signal, encoding='latin-1', *args, **kwds)
```


```python
hyperspy.io_plugins.ripple.parse_ripple(fp)
```

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 (Å)
  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
Parameters
  • filename (str) – the filename, either with the extension or without it
  • record_by (str) – ‘vector’ or ‘image’

hyperspy.io_plugins.ripple.write_rpl (filename, keys_dictionary, encoding='ascii')

hyperspy.io_plugins.semper_unf module

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

HEADERS = [("NCOL", '<i2'), ('NROW', '<i2'), ('NLAY', '<i2'), ('ICLASS', '<i2'), ('IFORM', '<i2'), ('IFLAG', '<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)), ('NLAY', ('<i2', 2)), ('ICOLN', ('<i2', 2)), ('NTITLE', '<i2'), ('TITLE', ('<i2', 144)), ('XUNIT', ('<i2', 4)), ('YUNIT', ('<i2', 4)), ('ZUNIT', ('<i2', 4))]

classmethod _check_format(data)

_convert_date_time_from_label()

_get_label()

classmethod _read_label(unf_file)

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** *(bool, 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._read_data` *(fobj, fname, position, data_format, shape)*

`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.sur module**

class `hyperspy.io_plugins.sur.DigitalSurfHandler` *(filename=None)*

Bases: object

Class to read Digital Surf MountainsMap files.

`filename, signal_dict, _work_dict, _list_sur_file_content, _Object_type, _N_data_object, _N_data_channels, _initialized`

parse_file, parse_header, get_image_dictionaries

Class Variables

--------------

_object_type : dict key: int containing the mountainsmap object types

_MS_parse *(strMS, prefix, delimiter)*

Parses a string containing metadata information. The string can be read from the comment section of a .sur file, or, alternatively, a file containing them with a similar formatting.

Parameters

• **strMS**(string containing metadata)–

• **prefix** *(string (or char) character assumed to start each line.)–

• **if a .sur file. ('$')–

• **delimiter** *(string that delimits the keyword from value. always '=')*–

Returns dictMS

Return type  dictionary in the correct hyperspy metadata format

3.1. hyperspy package
_append_work_dict_to_content()

Save the values stored in the work dict in the surface file list

_build_1D_series()

Build a series of 1D objects. The T axis is navigation and set from the first object

_build_RGB_image()

Build an RGB image. The T axis is navigation and set from P Size

_build_RGB_surface()

Build a series of surfaces. The T axis is navigation and set from P Size

_build_Tax(unpacked_dict, size_key, ind=0, nav=True)

Return T axis dictionary from an unpacked surface object dict. Unlike x and y axes, the size key can be determined from various keys: _14_W_Size, _15_Size_of_Points or _03_Number_of_Objects. index int and navigate boolean can be optionally passed. Default 0 and True respectively.

_build_Xax(unpacked_dict, ind=0, nav=False)

Return X axis dictionary from an unpacked dict. index int and navigate boolean can be optionally passed. Default 0 and False respectively.

_build_Yax(unpacked_dict, ind=1, nav=False)

Return X axis dictionary from an unpacked dict. index int and navigate boolean can be optionally passed. Default 1 and False respectively.

_build_general_1D_data()

Build general 1D Data objects. Currently work with spectra

_build_hyperspectral_map()

Build a hyperspectral map. Hyperspectral maps are single-object files with datapoints of _14_W_Size length

_build_metadata(unpacked_dict)

Return a minimalistic metadata dictionary according to hyperspy format. Accept a dictionary as an input because dictionary with the headers of a mountians object.

Parameters unpacked_dict (dictionary from the header of a surface file)

Returns metadict

Return type dictionary in the hyperspy metadata format

_build_original_metadata()

Builds a metadata dictionary from the header

_build_spectrum()

Build spectra objects. Spectra and 1D series of spectra are saved in the same object.

_build_sur_dict()

Create a signal dict with an unpacked object

_build_surface()

Build a surface

_build_surface_series()

Build a series of surfaces. The T axis is navigation and set from the first object

_check_comments(commentsstr, prefix, delimiter)

Check if comment string is parsable into metadata dictionary. Some specific lines (empty or starting with @@) will be ignored, but any non-ignored line must conform to being a title line (beginning with the TITLESTART indicator) or being parsable (starting with Prefix and containing the key data delimiter). At
the end, the comment is considered parsable if it contains minimum 1 parsable line and no non-ignorable non-parsable non-title line.

**Parameters**

- **commentstr** *(string containing comments)*
- **prefix** *(string (or char) character assumed to start each line.)*
- **if a .sur file. (\$)*
- **delimiter** *(string that delimits the keyword from value. always \'=\')*

**Returns** valid

**Return type** boolean

- `_get_bytes` *(file, size, default=None)*
- `_get_float` *(file, default=None)*
  Read a 4-bytes (single precision) float from a binary file f with a default value if no file is given
- `_get_int16` *(file, default=None, signed=True)*
  Read a 16-bits int with a user-definable default value if no file is given
- `_get_int32` *(file, default=None)*
  Read a 32-bits int with a user-definable default value if no file is given
- `_get_str` *(file, size, default=None, encoding='latin-1')*
  Read a str of defined size in bytes with a user-definable default value if no file is given
- `_get_uint32` *(file, default=None)*
- `_get_work_dict_key_value` *(key)*
- `_mountains_object_types = {-1: '_ERROR', 0: '_UNKNOWN', 1: '_PROFILE', 2: '_SURFACE', ...}_`
- `_pack_comment` *(file, val, encoding='latin-1')*
- `_pack_data` *(file, val, encoding='latin-1')*
  This needs to be special because it writes until the end of file.
- `_pack_private` *(file, val, encoding='latin-1')*
- `_read_single_sur_object` *(file)*
- `_read_sur_file` *
  Read the binary, possibly compressed, content of the surface file. Surface files can be encoded as single or a succession of objects. The file is thus read iteratively and from metadata of the first file
- `_set_bytes` *(file, val, size)*
- `_set_float` *(val)*
  write a 4-bytes (single precision) float in a file
- `_set_int16` *(file, val)*
- `_set_int32` *(file, val)*
  Write a 32-bits int in a file f
- `_set_metadata_and_original_metadata` *(unpacked_dict)*
  Run successively _build_metadata and _build_original_metadata and set signal dictionary with results
- `_set_str` *(file, val, size, encoding='latin-1')*
  Write a str of defined size in bytes to a file. struct.pack will automatically trim the string if it is too long

---

### 3.1. hyperspy package

443
This needs to be special because it reads until the end of file. This causes an error in the series of data

post_process_RGB(signal)

hyperspy.io_plugins.sur.file_reader(filename, **kwds)
Read a mountainsmap .sur file and return a dictionary containing the information necessary for creating the data object

Parameters filename (name of the .sur file to be read) –

Returns

• signal_dict (dictionary in the appropriate format. The dictionary can)
• contain several keys including ‘data’, ‘axes’, ‘metadata’, ‘original_metadata’,
• ‘post_process’, ‘mapping’, ‘attributes’.

hyperspy.io_plugins.tiff module

hyperspy.io_plugins.tiff._get_dm_kwargs_extratag(signal, scales, units, offsets)

hyperspy.io_plugins.tiff._get_imagej_kwargs(signal, scales, units, factor=100000000)

hyperspy.io_plugins.tiff._get_scale_unit(signal, encoding=None)
Return a list of scales and units, the length of the list is equal to the signal dimension.

hyperspy.io_plugins.tiff._get_scales_from_x_y_resolution(op, factor=1)

hyperspy.io_plugins.tiff._get_tags_dict(signal, extratags=[], factor=100000000)
Get the tags to export the scale and the unit to be used in Digital Micrograph and ImageJ.

hyperspy.io_plugins.tiff._imagej_description(version='1.11a', **kwargs)
Return a string that will be used by ImageJ to read the unit when appropriate arguments are provided

hyperspy.io_plugins.tiff._load_data(TF, filename, is_rgb, sl=None, memmap=None, **kwds)

hyperspy.io_plugins.tiff._parse_beam_current_FEI(value)

hyperspy.io_plugins.tiff._parse_scale_unit(tiff, op, shape, force_read_resolution)

hyperspy.io_plugins.tiff._parse_string(value)

hyperspy.io_plugins.tiff._parse_tuple_Zeiss(tup)

hyperspy.io_plugins.tiff._parse_tuple_Zeiss_with_units(tup, to_units=None)

hyperspy.io_plugins.tiff._parse_tvips_date(value)

hyperspy.io_plugins.tiff._parse_tvips_time(value)

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 tiff file 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**

**hyperspy.io_plugins.usid_hdf5 module**

**Module contents**

**hyperspy.learn package**

**Submodules**

**hyperspy.learn.mlPCA module**

**hyperspy.learn.mlPCA** *(X, varX, output_dimension, svd_solver='auto', tol=1e-10, max_iter=50000, **kwargs)*

Performs maximum likelihood PCA with missing data and/or heteroskedastic noise.

Standard PCA based on a singular value decomposition (SVD) approach assumes that the data is corrupted with Gaussian, or homoskedastic noise. For many applications, this assumption does not hold. For example, count data from EDS-TEM experiments is corrupted by Poisson noise, where the noise variance depends on the underlying pixel value. Rather than scaling or transforming the data to approximately “normalize” the noise, MLPCA instead uses estimates of the data variance to perform the decomposition.

This function is a transcription of a MATLAB code obtained from [Andrews1997].

Read more in the User Guide.

**Parameters**

• **X** *(numpy array, shape (m, n))* – Matrix of observations.

• **varX** *(numpy array)* – Matrix of variances associated with X (zeros for missing measurements).
• **output_dimension** *(int)* – The model dimensionality.

• **svd_solver** *("auto", "full", "arpack", "randomized"), default "auto")*–

  If auto: The solver is selected by a default policy based on *data.shape* and *output_dimension*: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

  If full: run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing

  If arpack: use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly $0 < \text{output_dimension} < \min(data.shape)$

  If randomized: use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

• **tol** *(float)* – Tolerance of the stopping condition.

• **max_iter** *(int)* – Maximum number of iterations before exiting without convergence.

Returns

• U, S, V *(numpy array)* – The pseudo-SVD parameters.

• s_obj *(float)* – Value of the objective function.

References

**hyperspy.learn.mva module**

class hyperspy.learn.mva.LearningResults

Bases: object

Stores the parameters and results from a decomposition.

   _transpose_results()

bss_algorithm = None

bss_factors = None

bss_loadings = None

centre = None

cluster_algorithm = None

cluster_centers = None

cluster_centers_estimated = None

cluster_labels = None

cluster_membership = None

cluster_metric = None

cluster_metric_data = None

cluster_metric_index = None
crop_decomposition_dimension \((n, \text{compute}=False)\)

Crop the score matrix up to the given number.

It is mainly useful to save memory and reduce the storage size

**Parameters**

- \(n\) (int) – Number of components to keep.
- compute (bool, default False) – If True and the decomposition results are lazy, also compute the results.

decomposition_algorithm = None

estimated_number_of_clusters = None

explained_variance = None

explained_variance_ratio = None

factors = None

load (filename)

Load the results of a previous decomposition and demixing analysis.

**Parameters**

filename (string) – Path to load the results from.

loadings = None

mean = None

navigation_mask = None

number_of_clusters = 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) – Path to save the results to.
- overwrite ([True, False, None], default None) – If True, overwrite the file if it exists. If None (default), prompt user if file exists.

signal_mask = None

summary()

Summarize the decomposition and demixing parameters.

**Returns**

String summarizing the learning parameters.

**Return type**

str

unfolded = None

unmixing_matrix = None
class hyperspy.learn.mva.MVA
    Bases: object

    Multivariate analysis capabilities for the Signal1D class.

    _auto_reverse_bss_component (reverse_component_criterion)
    _calculate_recmatrix (components=None, mva_type='decomposition')
        Rebuilds data from selected components.

        Parameters
        • components (None, int, or list of ints) –
          - If None, rebuilds signal instance from all components
          - If int, rebuilds signal instance from components in range 0-given int
          - If list of ints, rebuilds signal instance from only components in given list
        • mva_type (str {'decomposition', 'bss'}) – Decomposition type (not case sensitive)

        Returns Data built from the given components.
        Return type Signal instance

    _cluster_analysis (scaled_data, algorithm)
        Cluster analysis of a scaled data - internal

        Parameters
        • n_clusters (int) – Number of clusters to find.
        • scaled_data (numpy array - (number_of_samples, number_of_features)) –
        • algorithm (scikit learn clustering object) –
        • **kwargs – Additional parameters passed to the clustering algorithm. This may include
          n_init, the number of times the algorithm is restarted to optimize results.

        Returns return the sklearn.cluster object
        Return type alg

    _distances_within_cluster (cluster_data, memberships, squared=True, summed=False)
        Return inter cluster distances.

        Parameters
        • cluster_data (ndarray) – scaled cluster data
        • memberships (ndarray) – cluster labels
        • squared (bool, optional) – square distance measurement. The default is True.
        • summed (bool, optional) – If False returns array showing sum of distance from a
          given point to all other points in the cluster. If True returns a sum of all distances within a
          cluster. The results are scaled by 2*number of cluster points. The default is False.

        Returns result – list of distances for within the cluster
        Return type list

    _get_cluster_algorithm (algorithm, **kwargs)
        Convenience method to lookup cluster algorithm if algorithm is a string and instantiates it with n_clusters
        or if it’s an object check that the object has a fit method
_get_cluster_preprocessing_algorithm(algorithm, **kwargs)
Convenience method to lookup method if algorithm is a string or if it's an object check that the object has a fit_transform method

_get_cluster_signal(cluster_source, number_of_components=None, navigation_mask=None, signal_mask=None)
A cluster source can be an external signal, the signal data or the decomposition or bss results Return a flatten version of the data, nav and signal mask

Parameters

- **cluster_source** *(str or BaseSignal) – “decomposition”, “bss”, “signal” or a Signal*
- **number_of_components** *(int, optional) – Number of components to use with decomposition sources. The default is None.*
- **navigation_mask** *(ndarray, optional) – mask used to select regions of the cluster_source to use. The default is None.*
- **signal_mask** *(ndarray, optional) – mask used to select regions of the cluster_source signal. For decomposition or bss this is not used. The default is None.*
- **reproject** *(bool, optional) – If False the and the cluster_source is decomposition or bss the loadings are returned. If True the factor @ loadings result is used. The default is False.*

Returns toreturn – Returns an unfolded dataset from the selected cluster_source

Return type ndarray

_get_number_of_components_for_clustering()
Returns the number of components

_mask_for_clustering(mask)

_scale_data_for_clustering(cluster_signal, preprocessing='norm', preprocessing_kwargs={})
Scale data for cluster analysis
Results are stored in learning_results.

Parameters

- **cluster_signal** *("bss", "decomposition", "signal", Signal) – If “bss” the blind source separation results are used If “decomposition” the decomposition results are used if “signal” the signal data is used (signal should be unfolded)*
- **preprocessing** *("standard", "norm", "minmax", None or scikit learn preprocessing method) – default: ‘norm’ Preprocessing the data before cluster analysis requires preprocessing the data to be clustered to similar scales. Standard preprocessing adjusts each feature to have uniform variation. Norm preprocessing adjusts treats the set of features like a vector and each measurement is scaled to length 1. You can also pass a cikit-learn preprocessing object See scaling methods in scikit-learn preprocessing for further details.*
- **preprocessing_kwargs** *(Additional parameters passed to the cluster preprocessing algorithm. See sklearn.preprocessing preprocessing methods for further details)*

See also:

- clusters_analysis()
- estimate_number_of_clusters()
• get_cluster_labels(),
• get_cluster_signals(),
• plot_cluster_metric(),
• plot_cluster_results()
• plot_cluster_signals()
• plot_cluster_labels()

Returns
• scaled_data (numpy array - unfolded array of shape (number_of_samples,)
  no_of_features) scaled according to the selected algorithm

_unmix_components (compute=False)
blind_source_separation (number_of_components=None, algorithm='sklearn_fastica',
diff_order=1, diff_axes=None, factors=None,
comp_list=None, mask=None, on_loadings=False, reverse_component_criterion='factors',
whiten_method='PCA', return_info=False, print_info=True, **kwargs)
Apply blind source separation (BSS) to the result of a decomposition.

The results are stored in self.learning_results.

Read more in the User Guide.

Parameters

• number_of_components (int or None) – Number of principal components to
  pass to the BSS algorithm. If None, you must specify the comp_list argument.

• algorithm ("sklearn_fastica", "orthomax", "FastICA", "JADE", "CuBICA", "TDSEP", custom object), default
  "sklearn_fastica") – The BSS algorithm to use. If algorithm is an object,
  it must implement a fit_transform() method or fit() and transform() methods,
  in the same manner as a scikit-learn estimator.

• diff_order (int, default 1) – Sometimes it is convenient to perform the BSS
  on the derivative of the signal. If diff_order is 0, the signal is not differentiated.

• diff_axes (None or list of ints or strings) –
  – If None and on_loadings is False, when diff_order is greater than 1 and signal_dimension
    is greater than 1, the differences are calculated across all signal axes
  – If None and on_loadings is True, when diff_order is greater than 1 and navigation_dimension
    is greater than 1, the differences are calculated across all navigation axes
  – Otherwise the axes can be specified in a list.

• factors (BaseSignal or numpy 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 (None or list or numpy array) – Choose the components to apply BSS to. Unlike
  number_of_components, this argument permits non-contiguous components.
• **mask** (*BaseSignal* or subclass) – If not None, the signal locations marked as True are masked. The mask shape must be equal to the signal shape (navigation shape) when `on_loadings` is False (True).

• **on_loadings** (*bool*, *default* False) – If True, perform the BSS on the loadings of a previous decomposition, otherwise, perform the BSS on the factors.

• **reverse_component_criterion** ("factors", "loadings", *default* "factors") – Use either the factors or the loadings to determine if the component needs to be reversed.

• **whiten_method** ("PCA", "ZCA", None, *default* "PCA") – How to whiten the data prior to blind source separation. If None, no whitening is applied. See `whiten_data()` for more details.

• **return_info** (*bool*, *default* False) – The result of the decomposition is stored internally. However, some algorithms generate some extra information that is not stored. If True, return any extra information if available. In the case of sklearn.decomposition objects, this includes the sklearn Estimator object.

• **print_info** (*bool*, *default* True) – If True, print information about the decomposition being performed. In the case of sklearn.decomposition objects, this includes the values of all arguments of the chosen sklearn algorithm.

• ****kwargs** (*extra keyword arguments*) – Any keyword arguments are passed to the BSS algorithm.

**Returns**

return_info –

• If True and ‘algorithm’ is an sklearn Estimator, returns the Estimator object.

• Otherwise, returns None

**Return type** sklearn.Estimator or None

See also:

• `plot_bss_factors()`
• `plot_bss_loadings()`
• `plot_bss_results()`

**cluster_analysis**(*cluster_source*, *source_for_centers=None*, *preprocessing=None*, *preprocessing_kwargs={}*, *number_of_components=None*, *navigation_mask=None*, *signal_mask=None*, *algorithm=None*, *return_info=False*, **kwargs)

Cluster analysis of a signal or decomposition results of a signal. Results are stored in `learning_results`.

**Parameters**

• **cluster_source** ("bss", "decomposition", "signal", *BaseSignal*) – If “bss” the blind source separation results are used If “decomposition” the decomposition results are used if “signal” the signal data is used Note that using the signal or BaseSignal can be memory intensive and is only recommended if the Signal dimension is small BaseSignal must have the same navigation dimensions as the signal.

• **source_for_centers** (None, "decomposition", "bss", "signal", *BaseSignal*),) – default : None If None the cluster_source is used If “bss” the blind source separation results are used If “decomposition” the decomposition results are used
if “signal” the signal data is used BaseSignal must have the same navigation dimensions as the signal.

- **preprocessing** ("standard", "norm", "minmax", None or scikit learn preprocessing method) – default: ‘norm’ Preprocessing the data before cluster analysis requires preprocessing the data to be clustered to similar scales. Standard preprocessing adjusts each feature to have uniform variation. Norm preprocessing adjusts the set of features like a vector and each measurement is scaled to length 1. You can also pass one of the scikit-learn preprocessing scale_method = import sklearn.preprocessing.StandadScaler() preprocessing = scale_method See preprocessing methods in scikit-learn preprocessing for further details.

- **preprocessing_kwargs** (dict) – Additional parameters passed to the supported sklearn preprocessing methods. See sklearn.preprocessing scaling methods for further details

- **number_of_components** (int, default None) – If you are getting the cluster centers using the decomposition results (cluster_source_for_centers="decomposition") you can define how many components to use. If set to None the method uses the estimate of significant components found in the decomposition step using the elbow method and stored in the learning_results.number_significant_components attribute. This applies to both bss and decomposition results.

- **navigation_mask** (boolean numpy array) – The navigation locations marked as True are not used.

- **signal_mask** (boolean numpy array) – The signal locations marked as True are not used in the clustering for “signal” or Signals supplied as cluster source. This is not applied to decomposition results or source_for_centers (as it may be a different shape to the cluster source)

- **algorithm** ({ "kmeans" | "agglomerative" | "minibatchkmeans" | "spectralclustering"}) – See scikit-learn documentation. Default “kmeans”

- **return_info** (bool, default False) – The result of the cluster analysis is stored internally. However, the cluster class used contain a number of attributes. If True (the default is False) return the cluster object so the attributes can be accessed.

- ****kwargs** (dict optional, default - empty) – Additional parameters passed to the clustering class for initialization. For example, in case of the “kmeans” algorithm, n_init can be used to define the number of times the algorithm is restarted to optimize results.

Other Parameters **n_clusters** (int) – Number of clusters to find using the one of the predefined methods “kmeans”,”agglomerative”,”minibatchkmeans”,”spectralclustering” See sklearn.cluster for details

See also:

- estimate_number_of_clusters()
- get_cluster_labels()
- get_cluster_signals()
- get_cluster_distances()
- plot_cluster_metric()
- plot_cluster_results()
- plot_cluster_signals()
• `plot_cluster_labels()`

Returns

• If ‘return_info’ is True returns the Scikit-learn cluster object
• used for clustering. Useful if you wish to
• examine inertia or other outputs.

`decomposition(normalize_poissonian_noise=False, algorithm='SVD', output_dimension=None, center=None, auto_transpose=True, navigation_mask=None, signal_mask=None, var_array=None, var_func=None, reproject=None, return_info=False, print_info=True, svd_solver='auto', copy=True, **kwargs)`

Apply a decomposition to a dataset with a choice of algorithms.

The results are stored in `self.learning_results`.

Read more in the User Guide.

Parameters

• `normalize_poissonian_noise (bool, default False)` – If True, scale the signal to normalize Poissonian noise using the approach described in [Keenan2004].

• `algorithm ("SVD", "MLPCA", "sklearn_pca", "NMF", "sparse_pca", "mini_batch_sparse_pca", "RPCA", "ORPCA", "ORNMF", custom object), default "SVD")` – The decomposition algorithm to use. If algorithm is an object, it must implement a `fit_transform()` method or `fit()` and `transform()` methods, in the same manner as a scikit-learn estimator.

• `output_dimension (None or int)` – Number of components to keep/calculate. Default is None, i.e. `min(data.shape)`.

• `centre (None, "navigation", "signal"), default None)` –
  – If None, the data is not centered prior to decomposition.
  – If “navigation”, the data is centered along the navigation axis. Only used by the “SVD” algorithm.
  – If “signal”, the data is centered along the signal axis. Only used by the “SVD” algorithm.

• `auto_transpose (bool, default True)` – If True, automatically transposes the data to boost performance. Only used by the “SVD” algorithm.

• `navigation_mask (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. Only used by the “MLPCA” algorithm.

• `var_func (None or function or numpy array, default None)` –
  – If None, ignored
  – If function, applies the function to the data to obtain `var_array`. Only used by the “MLPCA” algorithm.
  – If numpy array, creates `var_array` by applying a polynomial function defined by the array of coefficients to the data. Only used by the “MLPCA” algorithm.
• **reproject** ((None, "signal", "navigation", "both"), default None) – If not None, the results of the decomposition will be projected in the selected masked area.

• **return_info** (bool, default False) – The result of the decomposition is stored internally. However, some algorithms generate some extra information that is not stored. If True, return any extra information if available. In the case of sklearn.decomposition objects, this includes the sklearn Estimator object.

• **print_info** (bool, default True) – If True, print information about the decomposition being performed. In the case of sklearn.decomposition objects, this includes the values of all arguments of the chosen sklearn algorithm.

• **svd_solver** ("auto", "full", "arpack", "randomized"), default "auto") –

  If auto: The solver is selected by a default policy based on data.shape and output_dimension: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

  If full: run exact SVD, calling the standard LAPACK solver via scipy.linalg.svd(), and select the components by postprocessing

  If arpack: use truncated SVD, calling ARPACK solver via scipy.sparse.linalg.svds(). It requires strictly 0 < output_dimension < min(data.shape)

  If randomized: use truncated SVD, calling sklearn.utils.extmath.randomized_svd() to estimate a limited number of components

• **copy** (bool, default True) –

  – If True, stores a copy of the data before any pre-treatments such as normalization in s._data_before_treatments. The original data can then be restored by calling s.undo_treatments().

  – If False, no copy is made. This can be beneficial for memory usage, but care must be taken since data will be overwritten.

• ****kwargs** (extra keyword arguments) – Any keyword arguments are passed to the decomposition algorithm.

**Returns**

**return_info** –

• If True and ‘algorithm’ in [‘RPCA’, ‘ORPCA’, ‘ORNMF’], returns the low-rank (X) and sparse (E) matrices from robust PCA/NMF.

• If True and ‘algorithm’ is an sklearn Estimator, returns the Estimator object.

• Otherwise, returns None

**Return type** tuple(numpy array, numpy array) or sklearn.Estimator or None
References

See also:

- `plot_decomposition_factors()`
- `plot_decomposition_loadings()`
- `plot_decomposition_results()`
- `plot_explained_variance_ratio()`
- `decomposition()` for lazy signals

`estimate_elbow_position(explained_variance_ratio=None, log=True, max_points=20)`

Estimate the elbow position of a scree plot curve.

Used to estimate the number of significant components in a PCA variance ratio plot or other “elbow” type curves.

Find a line between first and last point on the scree plot. With a classic elbow scree plot, this line more or less defines a triangle. The elbow should be the point which is the furthest distance from this line. For more details, see [Satopää2011].

**Parameters**

- `explained_variance_ratio` ([`None`, `numpy array`]) – Explained variance ratio values that form the scree plot. If None, uses the `explained_variance_ratio` array stored in `s.learning_results`, so a decomposition must have been performed first.

- `max_points` (`int`) – Maximum number of points to consider in the calculation.

**Returns**

- `elbow_position` – Index of the elbow position in the input array. Due to zero-based indexing, the number of significant components is `elbow_position + 1`.

**Return type** `int`

References

See also:

- `get_explained_variance_ratio()`
- `plot_explained_variance_ratio()`

`estimate_number_of_clusters(cluster_source, max_clusters=10, preprocessing=None, preprocessing_kwargs={}, number_of_components=None, navigation_mask=None, signal_mask=None, algorithm=None, metric='gap', n_ref=4, **kwargs)`

Performs cluster analysis of a signal for cluster sizes ranging from `n_clusters` = 2 to `max_clusters` (default 12) Note that this can be a slow process for large datasets so please consider reducing `max_clusters` in this case. For each cluster it evaluates the silhouette score which is a metric of how well separated the clusters are. Maximima or peaks in the scores indicate good choices for cluster sizes.

**Parameters**

- `cluster_source` (["bss", "decomposition", "signal" or `Signal`]) – If “bss” the blind source separation results are used If “decomposition” the decomposition results are used if “signal” the signal data is used Note that using the signal can be
memory intensive and is only recommended if the Signal dimension is small. Input Signal must have the same navigation dimensions as the signal instance.

- **max_clusters** (*int, default 10*) – Max number of clusters to use. The method will scan from 2 to max_clusters.

- **preprocessing** ("standard", "norm", "minmax" or sklearn-like preprocessing object) – default: ‘norm’ Preprocessing the data before cluster analysis requires preprocessing the data to be clustered to similar scales. Standard preprocessing adjusts each feature to have uniform variation. Norm preprocessing adjusts the set of features like a vector and each measurement is scaled to length 1. You can also pass an instance of a sklearn preprocessing module. See preprocessing methods in scikit-learn preprocessing for further details.

- **preprocessing_kwargs** (*dict, default empty*) – Additional parameters passed to the cluster preprocessing algorithm. See sklearn.preprocessing preprocessing methods for further details

- **number_of_components** (*int, default None*) – If you are getting the cluster centers using the decomposition results (cluster_source_for_centers="decomposition") you can define how many PCA components to use. If set to None the method uses the estimate of significant components found in the decomposition step using the elbow method and stored in the `learning_results.number_significant_components` attribute.

- **navigation_mask** (*boolean numpy array, default: None*) – The navigation locations marked as True are not used in the clustering.

- **signal_mask** (*boolean numpy array, default: None*) – The signal locations marked as True are not used in the clustering. Applies to “signal” or Signal cluster sources only.

- **metric** ("elbow", "silhouette", "gap") default ‘gap’) – Use distance, silhouette analysis or gap statistics to estimate the optimal number of clusters. Gap is believed to be, overall, the best metric but it’s also the slowest. Elbow measures the distances between points in each cluster as an estimate of how well grouped they are and is the fastest metric. For elbow the optimal k is the knee or elbow point. For gap the optimal k is the first k gap(k) >= gap(k+1)-std_error For silhouette the optimal k will be one of the “maxima” found with this method

- **n_ref** (*int, default 4*) – Number of references to use in gap statistics method Gap statistics compares the results from clustering the data to clustering uniformly distributed data. As clustering has a random variation it is typically averaged n_ref times to get an statistical average

- ****kwargs** (*dict {} default empty*) – Parameters passed to the clustering algorithm.

Other Parameters **n_clusters** (*int*) – Number of clusters to find using the one of the predefined methods “kmeans”, “agglomerative”, “minbatchkmeans”, “spectralclustering” See sklearn.cluster for details

Returns **best_k** – Estimate of the best cluster size

Return type **int**

See also:

- cluster_analysis()
- get_cluster_labels()
• get_cluster_signals(),
• get_cluster_distances(),
• plot_cluster_metric(),
• plot_cluster_results()
• plot_cluster_signals()
• plot_cluster_labels()

global

get_bss_model (components=None, chunks='auto')
Generate model with the selected number of independent components.

Parameters components ({None, int, list of ints}, default None) –
  • If None, rebuilds signal instance from all components
  • If int, rebuilds signal instance from components in range 0-given int
  • If list of ints, rebuilds signal instance from only components in given list

Returns A model built from the given components.
Return type Signal instance

global

global

get_decomposition_model (components=None)
Generate model with the selected number of principal components.

Parameters components ({None, int, list of ints}, default None) –
  • If None, rebuilds signal instance from all components
  • If int, rebuilds signal instance from components in range 0-given int
  • If list of ints, rebuilds signal instance from only components in given list

Returns A model built from the given components.
Return type Signal instance

global

get_explained_variance_ratio ()
Return explained variance ratio of the PCA components as a Signal1D.

Read more in the User Guide.

Returns s – Explained variance ratio.
Return type Signal1D

See also:

• decomposition()
• plot_explained_variance_ratio()
• get_decomposition_loadings()
• get_decomposition_factors()

global

normalize_bss_components (target='factors', function=<function sum>)
Normalize BSS components.

Parameters

  • target ("factors", "loadings") – Normalize components based on the scale of either the factors or loadings.
• **function** (*numpy universal function, default np.sum*) – Each target component is divided by the output of `function(target)`. The function must return a scalar when operating on `numpy` arrays and must have an `axis` argument.

**normalize_decomposition_components** (*target='factors', function=<function sum>*)

Normalize decomposition components.

**Parameters**

• **target** (*"factors", "loadings")* – Normalize components based on the scale of either the factors or loadings.

• **function** (*numpy universal function, default np.sum*) – Each target component is divided by the output of `function(target)`. The function must return a scalar when operating on `numpy` arrays and must have an `axis` argument.

**normalize_poissonian_noise** (*navigation_mask=None, signal_mask=None*)

Normalize the signal under the assumption of Poisson noise.

Scales the signal using to “normalize” the Poisson data for subsequent decomposition analysis [Keenan2004].

**Parameters**

• **navigation_mask** (*{None, boolean numpy array}, default None*) – Optional mask applied in the navigation axis.

• **signal_mask** (*{None, boolean numpy array}, default None*) – Optional mask applied in the signal axis.

**plot_cluster_metric**()

Plot the cluster metrics calculated using `evaluate_number_of_clusters` method.

See also:

• `estimate_number_of_clusters()`,
• `cluster_analysis()`,
• `get_cluster_labels()`,
• `get_cluster_signals()`,
• `plot_cluster_results()`,
• `plot_cluster_signals()`
• `plot_cluster_labels()`

**plot_cumulative_explained_variance_ratio** (*n=50*)

Plot cumulative explained variance up to `n` principal components.

**Parameters** **n** (*int*) – Number of principal components to show.

**Returns** **ax** – Axes object containing the cumulative explained variance plot.

**Return type** *`matplotlib.axes`*

See also:

• `plot_explained_variance_ratio()`
plot_explained_variance_ratio(n=30, log=True, threshold=0, hline='auto', vline=False, xaxis_type='index', xaxis_labeling=None, signal_fmt=None, noise_fmt=None, fig=None, ax=None, **kwargs)

Plot the decomposition explained variance ratio vs index number.

This is commonly known as a scree plot.

Read more in the User Guide.

Parameters

- n (int or None) – Number of components to plot. If None, all components will be plot.
- log (bool, default True) – If True, the y axis uses a log scale.
- threshold (float or int) – Threshold used to determine how many components should be highlighted as signal (as opposed to noise). If a float (between 0 and 1), threshold will be interpreted as a cutoff value, defining the variance at which to draw a line showing the cutoff between signal and noise; the number of signal components will be automatically determined by the cutoff value. If an int, threshold is interpreted as the number of components to highlight as signal (and no cutoff line will be drawn).
- hline (auto, True, False) – Whether or not to draw a horizontal line illustrating the variance cutoff for signal/noise determination. Default is to draw the line at the value given in threshold (if it is a float) and not draw in the case threshold is an int, or not given. If True, (and threshold is an int), the line will be drawn through the last component defined as signal. If False, the line will not be drawn in any circumstance.
- vline (bool, default False) – Whether or not to draw a vertical line illustrating an estimate of the number of significant components. If True, the line will be drawn at the the knee or elbow position of the curve indicating the number of significant components. If False, the line will not be drawn in any circumstance.
- xaxis_type ('index', 'number') – Determines the type of labeling applied to the x-axis. If 'index', axis will be labeled starting at 0 (i.e. “pythonic index” labeling); if 'number', it will start at 1 (number labeling).
- xaxis_labeling (ordinal, cardinal, None) – Determines the format of the x-axis tick labels. If 'ordinal', “1st, 2nd, ...” will be used; if 'cardinal', “1, 2,...” will be used. If None, an appropriate default will be selected.
- signal_fmt (dict) – Dictionary of matplotlib formatting values for the signal components
- noise_fmt (dict) – Dictionary of matplotlib formatting values for the noise components
- fig (matplotlib figure 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()

Returns

ax – Axes object containing the scree plot

Return type

matplotlib.axes
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': 'x',
             's': 200,
             'c': 'green'})
```

See also:

- `decomposition()`
- `get_explained_variance_ratio()`
- `get_decomposition_loadings()`
- `get_decomposition_factors()`

```python
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
```

```python
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
```

```python
undo_treatments()
```
Undo Poisson noise normalization and other pre-treatments.

Only valid if calling `s.decomposition(..., copy=True)`. 

hyperspy.learn.mva._get_derivative (signal, diff_axes, diff_order)
  Calculate the derivative of a signal.

hyperspy.learn.mva._normalize_components (target, other, function=<function sum>)
  Normalize components according to a function.

hyperspy.learn.ornmf module

class hyperspy.learn.ornmf.ORNMF (rank, store_error=False, lambda1=1.0, kappa=1.0,
  method='PGD', subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None)

Bases: object

Performs Online Robust NMF with missing or corrupted data.

The ORNMF code is based on a transcription of the online proximal gradient descent (PGD) algorithm MATLAB code obtained from the authors of [Zhao2016]. It has been updated to also include L2-normalization cost function that is able to deal with sparse corruptions and/or outliers slightly faster (please see ORPCA implementation for details). A further modification has been made to allow for a changing subspace \( W \), where \( X \approx WH^T + E \) in the ORNMF framework.

Read more in the User Guide.

References

Creates Online Robust NMF instance that can learn a representation.

Parameters

- **rank** (*int*) – The rank of the representation (number of components/factors)
- **store_error** (*bool*, *default False*) – If True, stores the sparse error matrix.
- **lambda1** (*float*) – Nuclear norm regularization parameter.
- **kappa** (*float*) – Step-size for projection solver.
- **method** (*{'PGD', 'RobustPGD', 'MomentumSGD'}, default 'PGD'*) –
  - ‘PGD’ - Proximal gradient descent
  - ‘RobustPGD’ - Robust proximal gradient descent
  - ‘MomentumSGD’ - Stochastic gradient descent with momentum
- **subspace_learning_rate** (*float*) – Learning rate for the ‘MomentumSGD’ method. Should be a float > 0.0
- **subspace_momentum** (*float*) – Momentum parameter for ‘MomentumSGD’ method, should be a float between 0 and 1.
- **random_state** (*None or int or RandomState instance, default None*) – Used to initialize the subspace on the first iteration.

_setup (X)

_solve_W (A, B)

finish()
  Return the learnt factors and loadings.
**fit** *(X, batch_size=None)*

Learn NMF components from the data.

**Parameters**

- **X** (*numpy.ndarray, iterator*) – [n_samples x n_features] matrix of observations or an iterator that yields samples, each with n_features elements.
- **batch_size** (*{None, int}*) – If not None, learn the data in batches, each of batch_size samples or less.

**project** *(X, return_error=False)*

Project the learnt components on the data.

**Parameters**

- **X** (*numpy.ndarray, iterator*) – [n_samples x n_features] matrix of observations or an iterator that yields n_samples, each with n_features elements.
- **return_error** (*bool*) – If True, returns the sparse error matrix as well. Otherwise only the weights (loadings)

```python
hyperspy.learn.ornmf._mrdivide(B, A)
Solves xB = A as per Matlab.
```

```python
hyperspy.learn.ornmf._project(W)
```

```python
hyperspy.learn.ornmf._solveproj(v, W, lambda1, kappa=1, h=None, e=None, vmax=None)
```

```python
hyperspy.learn.ornmf._thresh(X, lambda1, vmax)
```

**Perform online, robust NMF on the data X.**

This is a wrapper function for the ORNMF class.

**Parameters**

- **X** (*numpy array*) – The [n_samples, n_features] input data.
- **rank** (*int*) – The rank of the representation (number of components/factors)
- **store_error** (*bool, default False*) – If True, stores the sparse error matrix.
- **project** (*bool, default False*) – If True, project the data X onto the learnt model.
- **batch_size** (*{None, int}, default None*) – If not None, learn the data in batches, each of batch_size samples or less.
- **lambda1** (*float*) – Nuclear norm regularization parameter.
- **kappa** (*float*) – Step-size for projection solver.
- **method** (*{'PGD', 'RobustPGD', 'MomentumSGD'}, default 'PGD'*) –
  - ‘PGD’ - Proximal gradient descent
  - ‘RobustPGD’ - Robust proximal gradient descent
  - ‘MomentumSGD’ - Stochastic gradient descent with momentum
- **subspace_learning_rate** (*float*) – Learning rate for the ‘MomentumSGD’ method. Should be a float > 0.0
• **subspace_momentum** (*float*) – Momentum parameter for ‘MomentumSGD’ method, should be a float between 0 and 1.

• **random_state** *(None or int or RandomState instance, default None)* – Used to initialize the subspace on the first iteration.

**Returns**

• **Xhat** (*numpy array*) – is the [n_features x n_samples] non-negative matrix Only returned if store_error is True.

• **Ehat** (*numpy array*) – is the [n_features x n_samples] sparse error matrix Only returned if store_error is True.

• **W** (*numpy array, shape [n_features, rank]*) – is the non-negative factors matrix

• **H** (*numpy array, shape [rank, n_samples]*) – is the non-negative loadings matrix

**hyperspy.learn.orthomax module**

**hyperspy.learn.orthomax.orthomax** *(A, gamma=1.0, tol=1.4901e-07, max_iter=256)*

Calculate orthogonal rotations for a matrix of factors or loadings from PCA.

When gamma=1.0, this is known as varimax rotation, which finds a rotation matrix W that maximizes the variance of the squared components of A @ W. The rotation matrix preserves orthogonality of the components.

Taken from metpy.

**Parameters**

• **A** (*numpy array*) – Input data to unmix

• **gamma** (*float*) – If gamma in range [0, 1], use SVD approach, otherwise solve with a sequence of bivariate rotations.

• **tol** (*float*) – Tolerance of the stopping condition.

• **max_iter** (*int*) – Maximum number of iterations before exiting without convergence.

**Returns**

• **B** (*numpy array*) – Rotated data matrix

• **W** (*numpy array*) – The unmixing matrix

**hyperspy.learn.rpca module**

**class hyperspy.learn.rpca.ORPCA** *(rank, store_error=False, lambda1=0.1, lambda2=1.0, method='BCD', init='qr', training_samples=10, subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None)*

**Bases**: object

Perform Online Robust PCA with missing or corrupted data.

The ORPCA code is based on a transcription of MATLAB code from [Feng2013]. It has been updated to include a new initialization method based on a QR decomposition of the first n “training” samples of the data. A stochastic gradient descent (SGD) solver is also implemented, along with a MomentumSGD solver for improved convergence and robustness with respect to local minima. More information about the gradient descent methods and choosing appropriate parameters can be found in [Ruder2016].

Read more in the **User Guide**.
References

Creates Online Robust PCA instance that can learn a representation.

Parameters

- **rank** (*int*) – The rank of the representation (number of components/factors)
- **store_error** (*bool, default False*) – If True, stores the sparse error matrix.
- **lambda1** (*float*) – Nuclear norm regularization parameter.
- **lambda2** (*float*) – Sparse error regularization parameter.
- **method** (*{'CF', 'BCD', 'SGD', 'MomentumSGD'}, default 'BCD'*) –
  - ‘CF’ - Closed-form solver
  - ‘BCD’ - Block-coordinate descent
  - ‘SGD’ - Stochastic gradient descent
  - ‘MomentumSGD’ - Stochastic gradient descent with momentum
- **init** (*{'qr', 'rand', np.ndarray}, default 'qr'*) –
  - ‘qr’ - QR-based initialization
  - ‘rand’ - Random initialization
  - np.ndarray if the shape [n_features x rank]
- **training_samples** (*int*) – Specifies the number of training samples to use in the ‘qr’ initialization.
- **subspace_learning_rate** (*float*) – Learning rate for the ‘SGD’ and ‘MomentumSGD’ methods. Should be a float > 0.0
- **subspace_momentum** (*float*) – Momentum parameter for ‘MomentumSGD’ method, should be a float between 0 and 1.
- **random_state** (*None or int or RandomState instance, default None*) – Used to initialize the subspace on the first iteration.

__initialize_subspace__(X)

Initialize the subspace estimate.

__setup__(X)

__solve_L__(A, B)

finish(**kwargs)

Return the learnt factors and loadings.

fit(X, batch_size=None)

Learn RPCA components from the data.

Parameters

- **X** (*numpy.ndarray, iterator*) – [n_samples x n_features] matrix of observations or an iterator that yields samples, each with n_features elements.
- **batch_size** (*{None, int}* ) – If not None, learn the data in batches, each of batch_size samples or less.

project(X, return_error=False)

Project the learnt components on the data.
Parameters

- **X** ({numpy.ndarray, iterator}) – [n_samples x n_features] matrix of observations or an iterator that yields n_samples, each with n_features elements.

- **return_error** (bool) – If True, returns the sparse error matrix as well. Otherwise only the weights (loadings)

```python
hyperspy.learn.rpca._soft_thresh(X, lambda1)
```

Soft-thresholding of array X.

```python
hyperspy.learn.rpca._solveproj(z, X, Id, lambda2, r=None, e=None)
```

```python
hyperspy.learn.rpca._updatecol(X, A, B, Id)
```

```python
hyperspy.learn.rpca.orpca(X, rank, store_error=False, project=False, batch_size=None, lambda1=0.1, lambda2=1.0, method='BCD', init='qr', training_samples=10, subspace_learning_rate=1.0, subspace_momentum=0.5, random_state=None, **kwargs)
```

Perform online, robust PCA on the data X.

This is a wrapper function for the ORPCA class.

Parameters

- **X** ({numpy array, iterator}) – [n_features x n_samples] matrix of observations or an iterator that yields samples, each with n_features elements.

- **rank** (int) – The rank of the representation (number of components/factors)

- **store_error** (bool, default False) – If True, stores the sparse error matrix.

- **project** (bool, default False) – If True, project the data X onto the learnt model.

- **batch_size** ((None, int), default None) – If not None, learn the data in batches, each of batch_size samples or less.

- **lambda1** (float) – Nuclear norm regularization parameter.

- **lambda2** (float) – Sparse error regularization parameter.

- **method** ({'CF', 'BCD', 'SGD', 'MomentumSGD'}, default 'BCD') –
  - ‘CF’ - Closed-form solver
  - ‘BCD’ - Block-coordinate descent
  - ‘SGD’ - Stochastic gradient descent
  - ‘MomentumSGD’ - Stochastic gradient descent with momentum

- **init** ({'qr', 'rand', np.ndarray}, default 'qr') –
  - ‘qr’ - QR-based initialization
  - ‘rand’ - Random initialization
  - np.ndarray if the shape [n_features x rank]

- **training_samples** (int) – Specifies the number of training samples to use in the ‘qr’ initialization.

- **subspace_learning_rate** (float) – Learning rate for the ‘SGD’ and ‘MomentumSGD’ methods. Should be a float > 0.0

- **subspace_momentum** (float) – Momentum parameter for ‘MomentumSGD’ method, should be a float between 0 and 1.
• `random_state` *(None or int or RandomState instance, default None)* – Used to initialize the subspace on the first iteration.

**Returns**

- If `project` is True, returns the low-rank factors and loadings only
- Otherwise, returns the low-rank and sparse error matrices, as well as the results of a singular value decomposition (SVD) applied to the low-rank matrix.

**Return type** `numpy arrays`

```
hyperspy.learn.rpca.rpca_godec(X, rank, lambda1=None, power=0, tol=0.001, maxiter=1000, random_state=None, **kwargs)
```

Perform Robust PCA with missing or corrupted data, using the GoDec algorithm.

Decomposes a matrix $Y = X + E$, where $X$ is low-rank and $E$ is a sparse error matrix. This algorithm is based on the Matlab code from [Zhou2011]. See code here: https://sites.google.com/site/godecomposition/matrix/artifact-1

Read more in the *User Guide*.

**Parameters**

- `X` *(numpy array, shape (n_features, n_samples))* – The matrix of observations.
- `rank` *(int)* – The model dimensionality.
- `lambda1` *(None or float)* – Regularization parameter. If None, set to $1 / \sqrt{\text{n_features}}$
- `power` *(int, default 0)* – The number of power iterations used in the initialization
- `tol` *(float, default 1e-3)* – Convergence tolerance
- `maxiter` *(int, default 1000)* – Maximum number of iterations
- `random_state` *(None or int or RandomState instance, default None)* – Used to initialize the subspace on the first iteration.

**Returns**

- `Xhat` *(numpy array, shape (n_features, n_samples))* – The low-rank matrix
- `Ehat` *(numpy array, shape (n_features, n_samples))* – The sparse error matrix
- `U, S, V` *(numpy arrays)* – The results of an SVD on `Xhat`

**References**

`hyperspy.learn.svd_pca` module

```
hyperspy.learn.svd_pca.svd_flip_signs(u, v, u_based_decision=True)
```

Sign correction to ensure deterministic output from SVD.

Adjusts the columns of $u$ and the rows of $v$ such that the loadings in the columns in $u$ that are largest in absolute value are always positive.

**Parameters**

- `$v$` *(u,)* – $u$ and $v$ are the outputs of a singular value decomposition.
HyperSpy Documentation, Release 1.6.0

- `u_based_decision (bool, default True)` – If True, use the columns of u as the basis for sign flipping. Otherwise, use the rows of v. The choice of which variable to base the decision on is generally algorithm dependent.

**Returns** u, v – Adjusted outputs with same dimensions as inputs.

**Return type** numpy array

`hyperspy.learn.svd_pca.svd_pca (data, output_dimension=None, svd_solver='auto', centre=None, auto_transpose=True, svd_flip=True, **kwargs)`

Perform PCA using singular value decomposition (SVD).

Read more in the *User Guide*.

**Parameters**

- `data (numpy array)` – MxN array of input data (M features, N samples)
- `output_dimension (None or int)` – Number of components to keep/calculate
- `svd_solver ({"auto", "full", "arpack", "randomized"}, default "auto")` –
  - If `auto`: The solver is selected by a default policy based on `data.shape` and `output_dimension`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.
  - If `full`: run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing
  - If `arpack`: use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly `0 < output_dimension < min(data.shape)`
  - If `randomized`: use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components
- `centre (None, "navigation", "signal"), default None)` –
  - If None, the data is not centered prior to decomposition.
  - If “navigation”, the data is centered along the navigation axis.
  - If “signal”, the data is centered along the signal axis.
- `auto_transpose (bool, default True)` – If True, automatically transposes the data to boost performance.
- `svd_flip (bool, default True)` – If True, adjusts the signs of the loadings and factors such that the loadings that are largest in absolute value are always positive. See `svd_flip()` for more details.

**Returns**

- `factors (numpy array)
- `loadings (numpy array)
- `explained_variance (numpy array)
- `mean (numpy array or None (if centre is None))

`hyperspy.learn.svd_pca.svd_solve (data, output_dimension=None, svd_solver='auto', svd_flip=True, u_based_decision=True, **kwargs)`

Apply singular value decomposition to input data.
Parameters

- **data** *(numpy array, shape (m, n)) – Input data array*
- **output_dimension** *(None or int) – Number of components to keep/calculate*
- **svd_solver** *("auto", "full", "arpack", "randomized"), default "auto") –*

  If **auto**: The solver is selected by a default policy based on `data.shape` and `output_dimension`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient “randomized” method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

  If **full**: run exact SVD, calling the standard LAPACK solver via `scipy.linalg.svd()`, and select the components by postprocessing

  If **arpack**: use truncated SVD, calling ARPACK solver via `scipy.sparse.linalg.svds()`. It requires strictly `0 < output_dimension < min(data.shape)`

  If **randomized**: use truncated SVD, calling `sklearn.utils.extmath.randomized_svd()` to estimate a limited number of components

- **svd_flip** *(bool, default True) – If True, adjusts the signs of the loadings and factors such that the loadings that are largest in absolute value are always positive. See `svd_flip()` for more details.*

- **u_based_decision** *(bool, default True) – If True, and svd_flip is True, use the columns of u as the basis for sign-flipping. Otherwise, use the rows of v. The choice of which variable to base the decision on is generally algorithm dependent.*

Returns **U, S, V** – Output of SVD such that `X = U*S*V.T`

Return type *numpy array*

**hyperspy.learn.whitening module**

**hyperspy.learn.whitening.whiten_data** *(X, centre=True, method='PCA', epsilon=1e-10)*

Centre and whiten the data X.

A whitening transformation is used to decorrelate the variables, such that the new covariance matrix of the whitened data is the identity matrix.

If X is a random vector with non-singular covariance matrix C, and W is a whitening matrix satisfying `W^T W = C^-1`, then the transformation `Y = W X` will yield a whitened random vector Y with unit diagonal covariance. In ZCA whitening, the matrix `W = C^-1/2`, while in PCA whitening, the matrix W is the eigensystem of C. More details can be found in [Kessy2015].

Parameters

- **X** *(numpy array, shape (m, n)) – The input data.*
- **centre** *(bool, default True) – If True, centre the data along the features axis. If False, do not centre the data.*
- **method** *("PCA", "ZCA") – How to whiten the data. The default is PCA whitening.*
- **epsilon** *(float) – Small floating-point value to avoid divide-by-zero errors.*

Returns

- **Y** *(numpy array, shape (m, n)) – The centred and whitened data.*
HyperSpy Documentation, Release 1.6.0

- **W (numpy array, shape (n, n))** – The whitening matrix.

References

Module contents

hyperspy.misc package

Subpackages

hyperspy.misc.eds package

Submodules

hyperspy.misc.eds.ffast_mac module

hyperspy.misc.eds.utils module

hyperspy.misc.eds.utils.

• **get_element_and_line** (xray_line)
  Returns the element name and line character for a particular X-ray line as a tuple.
  By example, if xray_line = ‘Mn_Ka’ this function returns (‘Mn’, ‘Ka’)

hyperspy.misc.eds.utils.

• **get_energy_xray_line** (xray_line)
  Returns the energy (in keV) associated with a given X-ray line.
  By example, if xray_line = ‘Mn_Ka’ this function returns 5.8987

hyperspy.misc.eds.utils.

• **get_xray_lines_family** (xray_line)
  Returns the family to which a particular X-ray line belongs.
  By example, if xray_line = ‘Mn_Ka’ this function returns ‘Mn_K’

hyperspy.misc.eds.utils.

• **parse_only_lines** (only_lines)

hyperspy.misc.eds.utils.

• **quantification_cliff_lorimer** (intensities, kfactors, absorption_correction, ref_index=0, ref_index2=1)
  Quantification using Cliff-Lorimer

  Parameters

  • **intensities (numpy.array)** – the intensities for each X-ray lines. The first axis should be the elements axis.

  • **absorption_correction (numpy.array)** – value between 0 and 1 in order to correct the intensities based on estimated absorption.

  • **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’]

  • **ref_index2** (ref_index,) – index of the elements that will be in the denominator. Should be non zeros if possible.

  Returns

  • **numpy.array containing the weight fraction with the same shape as intensities.**
hyperspy.misc.eds.utilsedx_cross_section_to_zeta(cross_sections, elements)
Convert a list of cross_sections in barns (b) to zeta-factors (kg/m²).

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/m².

Return type list of float

hyperspy.misc.eds.utilselectron_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/cm³. 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_abs_corr_cross_section(composition, number_of_atoms, take_off_angle, probe_area)`
Calculate absorption correction terms.

Parameters

- `number_of_atoms (list of signal)` – Stack of maps with number of atoms per pixel.
- `take_off_angle (float)` – X-ray take-off angle in degrees.

`hyperspy.misc.eds.utils.get_abs_corr_zeta(weight_percent, mass_thickness, take_off_angle)`
Calculate absorption correction terms.

Parameters

- `weight_percent (list of signal)` – Composition in weight percent.
- `mass_thickness (signal)` – Density-thickness map in kg/m^2
- `take_off_angle (float)` – X-ray take-off angle in degrees.

`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, absorption_correction=None, 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
• `numpy.array` containing the weight fraction with the same shape as intensities.

`hyperspy.misc.eds.utils.quantification_cross_section(intensities, cross_sections, dose, absorption_correction=None)`

Quantification using EDX cross sections Calculate the atomic composition and the number of atoms per pixel from the raw X-ray intensity

**Parameters**

- `intensity (numpy.ndarray)` – The integrated intensity for each X-ray line, where the first axis is the element axis.
- `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 \( i \times t \times N/A \), \( i \) the current, \( t \) the acquisition time, and \( N \) the number of electrons per 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, absorption_correction=None)`

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 \( i \times t \times N \), \( i \) 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 and mass thickness in kg/m².

`hyperspy.misc.eds.utils.take_off_angle(tilt_stage, azimuth_angle, elevation_angle, beta_tilt=0.0)`

Calculate the take-off-angle (TOA).

TOA is the angle with which the X-rays leave the surface towards the detector.

**Parameters**

- `alpha_tilt (float)` – The alpha-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 alpha tilt axis.
- `elevation_angle (float)` – The elevation of the detector in degrees.
• **beta_tilt** (**float**) – The beta-tilt of the stage in degrees. The sample is facing positive 90 in the azimuthal direction when positively tilted.

**Returns**

**take_off_angle** – In degrees.

**Return type** **float**.

**Examples**

```python
>>> hs.eds.take_off_angle(alpha_tilt=10., beta_tilt=0.
>>> azimuth_angle=45., elevation_angle=22.)
28.865971201155283
```

**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 = xray_lines_model(['Cu', 'Fe'], beam_energy=30)
>>> s.plot()
```

**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

```python
hyperspy.misc.ed.s.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**

```python
class hyperspy.misc.eels.base_gos.GOSBase
    Bases: object
    get_parametrized_energy_axis(k1, k2, n)
    get_parametrized_qaxis(k1, k2, 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.

```python
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', verify_certificate=True)
```

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 element. 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.
- **verify_certificate** (bool) – If True, verify the eelsdb website certificate and raise an error if it is invalid. If False, continue querying the database if the certificate is invalid. (This is a potential security risk.)
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.electron_inelastic_mean_free_path module**

`hyperspy.misc.eels.electron_inelastic_mean_free_path._F(electron_energy)`

`hyperspy.misc.eels.electron_inelastic_mean_free_path._theta_E(density, electron_energy)`

`hyperspy.misc.eels.electron_inelastic_mean_free_path.iMFP_Iakoubovskii(density, electron_energy)`

Estimate electron inelastic mean free path from density

**density** [float] Material density in g/cm**3**

**beam_energy** [float] Electron beam energy in keV

For details see Equation 9 in: - Iakoubovskii, K., K. Mitsuishi, Y. Nakayama, and K. Furuya.


**Returns** Inelastic mean free path in nanometers

Return type `float`

`hyperspy.misc.eels.electron_inelastic_mean_free_path.iMFP_TPP2M(electron_energy, density, M, N_v, E_g)`

Electron inelastic mean free path using TPP-2M

**Parameters**

- `electron_energy (float)` – Electron beam energy in keV
• density (float) – Material density in g/cm**3
• M (float) – Molar mass in g/mol
• N_v (int) – Number of valence electron
• E_g (float) – Band gap in eV

Returns Inelastic mean free path in nanometers

Return type float

Notes


• hyperspy.misc.eels.electron_inelastic_mean_free_path.iMFP_angular_correction (density, beam_energy, alpha, beta)

Estimate the effect of limited collection angle on EELS mean free path

density [float] Material density in g/cm**3
beam_energy [float] Electron beam energy in keV
alpha, beta [float] Convergence and collection angles in mrad.


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 Strenght 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

_load_dictionary (dictionary)
_load_name = 'Hartree-Slater'

as_dictionary (fullcopy=True)
Export the GOS as a dictionary

integrateq (onset_energy, angle, E0)

readgosfile()

hyperspy.misc.eels.hydrogenic_gos module

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 (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.

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 Hydrogeninc 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

_name = 'hydrogenic'
gosfuncK (E, qa02)  
gosfuncL (E, qa02)  
integrateq (onset_energy, angle, E0)

hyperspy.misc.eels.tools module

hyperspy.misc.eels.tools._estimate_correlation_factor (g0, gk, k)

hyperspy.misc.eels.tools._estimate_gain (ns, cs, weighted=False, higher_than=None, plot_results=False, binning=0, pol_order=1)

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.

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

Parameters

- **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.

- **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.

Returns k

Return type Signal instance

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.

3.1. hyperspy package 479
Parameters

- **clean_SI** *(noisy_SI,)*
- **mask** *(numpy bool array)* – To define the channels that will be used in the calculation.
- **pol_order** *(int)* – The order of the polynomial.
- **higher_than** *(float)* – To restrict the fit to counts over the given value.
- **return_results** *(Bool)*
- **plot_results** *(Bool)*
- **store_results** *(True, False, "ask"), default "ask")* – If True, it stores the result in the signal metadata.

Returns

- *Dictionary with the result of a linear fit to estimate the offset and scale factor*

**hyperspy.misc.eels.tools.get_edges_near_energy** *(energy, width=10, only_major=False, order='closest')*

Find edges near a given energy that are within the given energy window.

Parameters

- **energy** *(float)* – Energy to search, in eV
- **width** *(float)* – Width of window, in eV, around energy in which to find nearby energies, i.e. a value of 10 eV (the default) means to search +/- 5 eV. The default is 10.
- **only_major** *(bool)* – Whether to show only the major edges. The default is False.
- **order** *(str)* – Sort the edges, if ‘closest’, return in the order of energy difference, if ‘ascending’, return in ascending order, similarly for ‘descending’

Returns **edges** – All edges that are within the given energy window, sorted by energy difference to the given energy.

Return type **list**

**hyperspy.misc.eels.tools.get_info_from_edges** *(edges)*

Return the information of a sequence of edges as a list of dictionaries.

Parameters **edges** *(str or iterable)* – the sequence of edges, each entry in the format of ‘element_subshell’.

Returns **info** – a list of dictionaries with information corresponding to the provided edges.

Return type **list**

**hyperspy.misc.eels.tools.power_law_perc_area** *(E1, E2, r)*

**hyperspy.misc.eels.tools.ratio** *(edge_A, edge_B)*

**hyperspy.misc.eels.tools.rel_std_of_fraction** *(a, std_a, b, std_b, corr_factor=1)*
Module contents

hyperspy.misc.holography package

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`.

`hyperspy.misc.holography.reconstruct.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 (bool)` – Plots the masked sideband used for reconstruction.

Returns `wav` – Reconstructed electron wave

Return type  `nparray`

`hyperspy.misc.holography.tools module`

`hyperspy.misc.holography.tools.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`

`hyperspy.misc.holography.tools.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**

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._fill_array_with_stream(spectrum_image, stream, first_frame, last_frame, re-bin_energy=1)

hyperspy.misc.io.fei_stream_readers._fill_array_with_stream_sum_frames(spectrum_image, stream, first_frame, last_frame, re-bin_energy=1)

hyperspy.misc.io.fei_stream_readers._stream_to_sparse_COO_array(stream_data, last_frame, shape, channels, re-bin_energy=1, first_frame=0)
hyperspy.misc.io.fei_stream_readers.stream_to_sparse_COO_array_sum_frames(stream_data, last_frame, shape, channels, rebin_energy=1, first_frame=0)

hyperspy.misc.io.fei_stream_readers.array_to_stream(array)

Convert an array to a FEI stream

Parameters

array (array) –

hyperspy.misc.io.fei_stream_readers.stream_to_array(stream, spatial_shape, channels, last_frame, 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) – dttype 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.convert_xml_to_dict(xml_object)`

`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.tools.xml2dtb(et, dictree)`

**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’`.

3.1. hyperspy package
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.* and randomized_svd from scikit-learn

hyperspy.misc.machine_learning.tools module

hyperspy.misc.machine_learning.tools.amari(W, A)
    Calculate the Amari distance between two non-singular matrices.
    Convenient for checking convergence in ICA algorithms (See [Moreau1998] and [Bach2002]).

    Parameters
    • A (W,) – The two matrices to measure.

    Returns
    Amari distance between W and A.

    Return type
    float

References

Module contents

Submodules

hyperspy.misc.array_tools module

hyperspy.misc.array_tools.linear_bin(dat, scale, crop=True)
    Binning of the spectrum image by a non-integer pixel value.

    Parameters
    • originalSpectrum (numpy.array) –

    • scale (a list of floats) – For each dimension specify the new:old pixel ratio, e.g. a ratio of 1 is no binning; 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 dimensions of the data.

    • 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 optional to preserve total counts before and after binning.

**Returns** with new dimensions width/scale for each dimension in the data.

**Return type** np.array

```
hyperspy.misc.array_tools._linear_bin_loop(result, data, scale)
```

```
hyperspy.misc.array_tools._requires_linear_rebin(arr, scale)
```

Returns True if linear_rebin is required.

- **param arr**: numpy array to rebin
- **type arr**: array
- **param scale**: rebinning factors
- **type scale**: tuple

```
hyperspy.misc.array_tools.are_aligned(shape1, shape2)
```

Check if two numpy arrays are aligned.

- **Parameters**
  - **shape2 (shape1,)** -

- **Returns** isaligned

- **Return type** bool

```
hyperspy.misc.array_tools.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

```
hyperspy.misc.array_tools.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.

```
hyperspy.misc.array_tools.homogenize_ndim(*args)
```

Given any number of arrays returns the same arrays reshaped by adding facing dimensions of size 1.

```
hyperspy.misc.array_tools.numba_histogram(data, bins, ranges)
```

Parameters

- **data (numpy array)** – Input data. The histogram is computed over the flattened array.
- **bins (int)** – Number of bins
- **ranges ((float, float))** – The lower and upper range of the bins.

- **Returns** hist – The values of the histogram.

- **Return type** 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.

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

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
**hyperspy.misc.config_dir module**

**hyperspy.misc.date_time_tools module**

**hyperspy.misc.date_time_tools.ISO_format_to_serial_date**(date, time, time_zone='UTC')

Convert ISO format to a serial date.

**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
    General
    date = 1991-10-01
    original_filename = example1.msa
    time = 12:00:00
    title = NIO EELS OK SHELL

>>> 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.miss.elements module**

**hyperspy.miss.example_signals_loading module**

**hyperspy.miss.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.miss.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.miss.example_signals_loading.load_object_hologram()**

Load an object hologram image
Notes

Sample: Fe needle with YOx nanoparticle inclusions. See reference for more details


TEM: FEI Titan G2 60-300 HOLO


`hyperspy.misc.example_signals_loading.load_reference_hologram()`

Load a reference hologram image

Notes

Sample: Fe needle with YOx nanoparticle inclusions. See reference for more details


TEM: FEI Titan G2 60-300 HOLO


`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 fullcopy=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

```python
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
hyperspy.misc.export_dictionary.parse_flag_string(flags)
```

```python
hyperspy.misc.export_dictionary.reconstruct_object(flags, value)
```

Reconstructs the value (if necessary) after having saved it in a dictionary

**hyperspy.misc.hist_tools module**

```python
hyperspy.misc.hist_tools._freedman_bw_dask(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.

```python
hyperspy.misc.hist_tools._scott_bw_dask(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.

```python
hyperspy.misc.hist_tools.histogram(a, bins='fd', range=None, max_num_bins=250, weights=None, **kwargs)
```

Enhanced histogram.

This is a histogram function that enables the use of more sophisticated algorithms for determining bins.

**Parameters**

- `a (array_like)` – Input data. The histogram is computed over the flattened array.
- `bins (int or sequence of scalars or str, default "fd")` – If `bins` is an int, it defines the number of equal-width bins in the given range. If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If `bins` is a string from the list below, will use the method chosen to calculate the optimal bin width and consequently the number of bins (see **Notes** for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the ‘auto’ option is suggested. Weighted data is not supported for automated bin size selection.

- `'auto'` Maximum of the ‘sturges’ and ‘fd’ estimators. Provides good all around performance.
- `'fd'` (Freedman Diaconis Estimator) Robust (resilient to outliers) estimator that takes into account data variability and data size.
- `'doane'` An improved version of Sturges’ estimator that works better with non-normal datasets.
- `'scott'` Less robust estimator that that takes into account data variability and data size.
'stone’ Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott’s rule.

'rice’ Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

'sturges’ R’s default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.

'sqrt’ Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.

'knuth’ Knuth’s rule is a fixed-width, Bayesian approach to determining the optimal bin width of a histogram.

'blocks’ Determination of optimal adaptive-width histogram bins using the Bayesian Blocks algorithm.

- **range** ((float, float), optional) – The lower and upper range of the bins. If not provided, range is simply \((a.\text{min}(a), a.\text{max}(a))\). Values outside the range are ignored. The first element of the range must be less than or equal to the second. range affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within range, the bin count will fill the entire range including portions containing no data.

- **max_num_bins** (int, default 250) – When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a MemoryError being raised by numpy.histogram().

- **weights** (array_like, optional) – An array of weights, of the same shape as a. Each value in a only contributes its associated weight towards the bin count (instead of 1). This is currently not used by any of the bin estimators, but may be in the future.

- ****kwargs – Passed to numpy.histogram()

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()

hyperspy.misc.hist_tools.histogram_dask(a, bins='fd', max_num_bins=250, **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, default 10) – If bins is a string, then it must be one of:

  - 'fd' (Freedman Diaconis Estimator) Robust (resilient to outliers) estimator that takes into account data variability and data size.

  - 'scott' Less robust estimator that that takes into account data variability and data size.
• **max_num_bins** (*int*, default 250) – When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a MemoryError being raised by *numpy.histogram*().

• **kwargs** – Passed to *dask.histogram()*

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:

• *dask.histogram()*

• *numpy.histogram()*

### hyperspy.misc.ipython_tools module

**hyperspy.misc.ipython_tools.get_interactive_ns()**

**hyperspy.misc.ipython_tools.get_ipython()**

Get the global InteractiveShell instance.

Returns None if no InteractiveShell instance is registered.

**hyperspy.misc.ipython_tools.is_it_running_from_ipython()**

**hyperspy.misc.ipython_tools.turn_logging_off()**

**hyperspy.misc.ipython_tools.turn_logging_on(verbosel*)

### hyperspy.misc.label_position module

**class hyperspy.misc.label_position.SpectrumLabelPosition(signal)**

* Bases: object

A class to get the positions of labels in spectrums such as EELS, EDX, XRF. The main method is the *get_markers* method which the user supplies a dictionary of edge labels specifying its energy positions, and it returns markers for labelling them.

  • _check_signal_figure_changed()__

  • _element_colour_dict(edges)

  • _estimate_textbox_dimension(dummy_text='My_g8')

  • _get_bbox_from_textbox_patch(fig, textbox)

  • _get_current_signal_index()

  • _get_textbox_pos(edges, offset=None, step=None, lb=None, ub=None)

  • _set_active_figure_properties()

  • _text_parser(text_edge)

  • _unique_element_of_edges(edges)

  colour_list_label = ['black', 'darkblue', 'darkgreen', 'darkcyan', 'darkmagenta', 'dimgray', 'brown', ' deeppink', 'olive', 'crimson']

3.1. hyperspy package
```python
edge_label_style = {'bbox': {'alpha': 0.2, 'facecolor': 'white'}, 'ha': 'center', 'va': 'center'}

def get_markers(labels):
    # Get the markers (vertical line segment and text box) for labelling the edges

    Parameters:
    labels (dictionary) – A dictionary with the labels as keys and their energies as values. E.g. for EELS edges it could be {'Mn_L2': 651.0, 'Cr_L3': 575.0}.

    Returns:
    - vls (list) – A list contains HyperSpy’s vertical line segment marker
    - txs (list) – A list contains HyperSpy’s text marker

hyperspy.misc.material module

```python

def atomic_to_weight(atomic_percent, elements):
    # Convert atomic percent to weight percent.

    Parameters:
    - atomic_percent (array) – The atomic fractions (composition) of the sample.
    - elements (list of str) – A list of element abbreviations, e.g. ['Al', 'Zn']

    Returns:
    weight_percent – composition in weight percent.

    Return type
    array of float

    Examples

    Calculate the weight percent of modern bronze given its atomic percent:
    >>> hs.material.atomic_to_weight([93.2, 6.8], ('Cu', 'Sn'))
    array([ 88.00501989, 11.99498011])

```python

def density_of_mixture(weight_percent, elements, mean='harmonic'):
    # Calculate the density 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 (array) – 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']
    - 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']) 8.6903187973131466

hyperspy.misc.material._elements_auto(composition, elements)
hyperspy.misc.material._lines_auto(composition, xray_lines)
hyperspy.misc.material._mass_absorption_mixture(weight_percent, elements, energies)

Calculate the mass absorption coefficient for X-ray absorbed in a mixture of elements.

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** (np.array) – The composition of the absorber(s) in weight percent. The first dimension of the matrix corresponds to the elements.
- **elements** (list of str) – The list of element symbol of the absorber, e.g. [‘Al’, ‘Zn’].
- **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’.

Examples

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

Returns

- float or array of float
- mass absorption coefficient(s) in cm²/g

See also:

mass_absorption()


hyperspy.misc.material._weight_to_atomic(weight_percent, elements)

Convert weight percent (wt%) to atomic percent (at.%).

Parameters

- **weight_percent** (array of float) – The weight fractions (composition) of the sample.
- **elements** (list of str) – A list of element abbreviations, e.g. ['Al', 'Zn']

Returns

- atomic_percent (array of float) – Composition in atomic percent.

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])

hyperspy.misc.material.atomic_to_weight(atomic_percent, elements='auto')
Convert atomic percent to weight percent.

Parameters

• atomic_percent (list of float or list of signals) – The atomic 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 atomic_percent list.

Returns weight_percent – composition in weight percent.

Return type as atomic_percent

Examples

Calculate the weight percent of modern bronze given its atomic percent: >>> hs.material.atomic_to_weight([93.2, 6.8], ('Cu', 'Sn')) array([ 88.00501989, 11.99498011])

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']) 8.6903187973131466

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 cm^2/g

Examples

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

See also:

`mass_absorption_mixture()`


```python
hyperspy.misc.material.mass_absorption_mixture(weight_percent, elements='auto', energies='auto')
```

Calculate the mass absorption coefficient for X-ray absorbed in a mixture of elements.

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 cm^2/g`

See also:

`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])

hyperspy.misc.math_tools module

hyperspy.misc.math_tools.antisymmetrize(a)

hyperspy.misc.math_tools.anyfloatin(things)

Check if iterable contains any non integer.

hyperspy.misc.math_tools.check_random_state(seed, lazy=False)

Turn a random seed into a np.random.RandomState instance.

Parameters

- **seed** *(None or int or np.random.RandomState or dask.array.random.RandomState)* –
  
  If None: Return the RandomState singleton used by np.random or dask.array.random
  
  If int: Return a new RandomState instance seeded with seed.
  
  If np.random.RandomState: Return it.
  
  If dask.array.random.RandomState: Return it.

- **lazy** *(bool, default False)* – If True, and seed is None or int, return a

hyperspy.misc.math_tools.closest_nice_number(number)

hyperspy.misc.math_tools.get_linear_interpolation(p1,p2,x)

Given two points in 2D returns y for a given x for y = ax + b

Parameters

- **p1,p2 ((x, y)*) –
- **x (float) –

Returns y

Return type float

hyperspy.misc.math_tools.hann_window_nth_order(m, order)

Calculates 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` – window

**Return type**

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.optimal_fft_size** (*target, real=False*)

Wrapper around scipy function `next_fast_len()` for calculating optimal FFT padding.

scipy.fft was only added in 1.4.0, so we fall back to scipy.fftpack if it is not available. The main difference is that `next_fast_len()` does not take a second argument in the older implementation.

**Parameters**

- **target** (*int*) – Length to start searching from. Must be a positive integer.
- **real** (*bool, optional*) – True if the FFT involves real input or output, only available for scipy > 1.4.0

**Returns**

Optimal FFT size.

**Return type**

int

**hyperspy.misc.math_tools.order_of_magnitude** (*number*)

Order of magnitude of the given number

**Parameters**

`number` (*float*) –

**Returns**

**Return type**

Float

**hyperspy.misc.math_tools.outer_nd** (*vec*)

Calculates outer product of n vectors

**Parameters**

`vec` (*vector*) –

**Returns**

`out` –

**Return type**

ndarray

**hyperspy.misc.math_tools.symmetrize** (*a*)

**hyperspy.misc.model_tools module**

**hyperspy.misc.model_tools._is_iter** (*val*)

Checks if value is a list or tuple

**hyperspy.misc.model_tools._iter_join** (*val*)

Joins values of iterable parameters for the fancy view, unless it equals None, then blank

**hyperspy.misc.model_tools._non_iter** (*val*)

Returns formatted string for a value unless it equals None, then blank
class hyperspy.misc.model_tools.current_component_values(component, only_free=False, only_active=False)

Bases: object

Convenience class that makes use of __repr__ methods for nice printing in the notebook of the properties of parameters of a component

Parameters

- **component** (hyperspy component instance) –
- **only_free** (bool, default False) – If True: Only include the free parameters in the view
- **only_active** (bool, default False) – If True: Helper for current_model_values. Only include active components in the view. Always shows values if used on an individual component.

__repr_html__() class hyperspy.misc.model_tools.current_model_values(model, only_free=False, only_active=False, component_list=None)

Bases: object

Convenience class that makes use of __repr__ methods for nice printing in the notebook of the properties of parameters in components in a model

Parameters

- **component** (hyperspy component instance) –
- **only_free** (bool, default False) – If True: Only include the free parameters in the view
- **only_active** (bool, default False) – If True: Only include active parameters in the view

__repr_html__() hyperspy.misc.physical_constants module hyperspy.misc.physics_tools module

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 diffracted beam. This is)
- **two times the bragg angle.**

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**  
\[ Z\_list \text{ (list of tuples)} \] – 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**

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._get_shapes(am, ignore_axis)
hyperspy.misc.signal_tools.are_signals_aligned(*args, ignore_axis=None)
hyperspy.misc.signal_tools.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
Bases: object

_get_array_slices(slices, isNavigation=None)
_slicer(slices, isNavigation=None, out=None)
**class** hyperspy.misc.slicing.SpecialSlicers(obj, isNavigation)

**Bases:** object

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._slice_target**(target, dims, both_slices, slice_nav=None, issignal=False)

Slices the target if appropriate

**Parameters**

- **target** *(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
- **slice_nav** *(bool, None)* – if None, target is returned as-is. Otherwise navigation and signal dimensions are sliced for True and False values respectively.
- **issignal** *(bool)* – if the target is signal and should be sliced as one

**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**

- **all_warnings()**
  
  Context for use in testing to ensure that all warnings are raised.

**Examples**

```python
>>> 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)
```

- **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 `numpy.testing.assert_allclose()` and checks all other values with `numpy.testing.assert_equal()`.
  Accepts additional positional and keyword arguments and pass those intact to `assert_allclose` (that’s how you specify comparison precision).

  **Parameters**
  
  - `actual` (list, dict or tuple) – Actual values to compare.
  - `expected` (list, dict or tuple) – Expected values.
  - `*args` – Arguments are passed to `numpy.testing.assert_allclose()` or `assert_deep_almost_equal()`.
  - `**kwargs` – Keyword arguments are passed to `numpy.testing.assert_allclose()` or `assert_deep_almost_equal()`.

- **assert_warns(message=None, category=None)**
  
  Context for use in testing to catch known warnings matching regexes.
  
  Allows for three types of behaviors: “and”, “or”, and “optional” matches. This is done to accomodate different build enviroments 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 `\A\Z` in a pattern to signify it as optional.

  **Parameters**
  
  - `message` (list of str or compiled regexes) – Regexes for the desired warning to catch.
  - `category` (list of type) – Warning categories for the desired warning to catch.

  **Raises** `ValueError` – If any match was not found or an unexpected warning was raised.
Examples

```python
>>> from skimage import data, img_as_ubyte, img_as_float
>>> with assert_warns(['precision loss']):
...     d = img_as_ubyte(img_as_float(data.coins()))
```

Notes

Upon exiting, it checks the recorded warnings for the desired matching pattern(s).

```python
hyperspy.misc.test_utils.check_closing_plot(s)
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**

```python
hyperspy.misc.tv_denoise._tv_denoise_1d(im, weight=50, eps=0.0002, keep_type=False, n_iter_max=200)
```

Perform total-variation denoising

**Parameters**

- **im** *(ndarray)* – input data to be denoised
- **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


This code is an implementation of the algorithm of Rudin, Fatemi and Osher that was proposed by Chambolle in\(^0\).

References

Examples

```python
>>> 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=60.0)
```

hypospy.miss.tv_denoise._tv_denoise_2d(im, weight=50, eps=0.0002, keep_type=\text{False}, n_iter_max=200)

Perform total-variation denoising

Parameters

- \text{im} (\text{ndarray}) \text{ – input data to be denoised}
- \text{weight} (\text{float, optional}) \text{ – denoising weight. The greater weight, the more denoising (at the expense of fidelity to input)}
- \text{eps} (\text{float, optional}) \text{ – relative difference of the value of the cost function that determines the stop criterion. The algorithm stops when:}
  \[(E_{(n-1)} - E_n) < \text{eps} \times E_0\]
- \text{keep_type} (\text{bool, optional (False)}) \text{ – 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}
- \text{n_iter_max} (\text{int, optional}) \text{ – maximal number of iterations used for the optimization.}

Returns \text{out} \text{ – denoised array}

Return type \text{ndarray}

Notes

The principle of total variation denoising is explained in http://en.wikipedia.org/wiki/Total_variation_denoising

This code is an implementation of the algorithm of Rudin, Fatemi and Osher that was proposed by Chambolle in\footnote{A. Chambolle, An algorithm for total variation minimization and applications, Journal of Mathematical Imaging and Vision, Springer, 2004, 20, 89-97.}

References

Examples

```python
>>> import scipy
>>> ascent = scipy.ascent().astype(np.float)
>>> ascent += 0.5 * ascent.std() * np.random.randn(*ascent.shape)
>>> denoised_ascent = tv_denoise(ascent, weight=60.0)
```

hypostry.miss.tv_denoise._tv_denoise_3d(im, weight=100, eps=0.0002, keep_type=\text{False}, n_iter_max=200)

Perform total-variation denoising on 3-D arrays

Parameters
• **im** (*ndarray*) – 3-D input data to be denoised

• **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:

\[
(E_{n-1} - E_n) < eps \times E_0
\]

• **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**

Rudin, Osher and Fatemi algorithm

**Examples**

First build synthetic noisy data

```python
>>> 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)
```

Perform total-variation denoising on 2-d and 3-d images

```python
hyperspy.misc.tv_denoise.tv_denoise(im, weight=50, eps=0.0002, keep_type=False, n_iter_max=200)
```

**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:

\[
(E_{n-1} - E_n) < eps \times E_0
\]

• **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.
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\(^0\).

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

```python
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 constains
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.
```


3.1. hyperspy package 509
Examples

```
>>> tree = DictionaryTreeBrowser()
>>> tree.set_item("Branch.Leaf1.color", "green")
>>> tree.set_item("Branch.Leaf2.color", "brown")
>>> tree.set_item("Branch.Leaf2.caterpillar", True)
>>> tree.set_item("Branch.Leaf1.caterpillar", False)

>>> tree

Branch
  Leaf1
    caterpillar = False
    color = green
  Leaf2
    caterpillar = True
    color = brown

>>> tree.Branch

Leaf1
  caterpillar = False
  color = green
Leaf2
  caterpillar = True
  color = brown

>>> for label, leaf in tree.Branch:
...    print("%s is %s" % (label, leaf.color))

Leaf1 is green
Leaf2 is brown

>>> tree.Branch.Leaf2.caterpillar
True

>>> "Leaf1" in tree.Branch
True

>>> "Leaf3" in tree.Branch
False
```

`_get_html_print_items(padding='', max_len=78, recursive_level=0)`
Recursive method that creates a html string for fancy display of metadata.

`_get_print_items(padding='', max_len=78)`
Prints only the attributes that are not methods

`_repr_html_()`

`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
```

```python
dict_browser
```

```python
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
```

text

**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_key_value(key, value)

Returns the metadata value as a html string

hyperspy.misc.utils.add_scalar_axis(signal, lazy=None)

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)

```python
self.signal.data = np.arange(10) + 2
```

The behaviour is identical to >>> self.signal.data = np.arange(10) + 2

 hyperspy.misc.utils.check_long_string (value, max_len)
 Checks whether string is too long for printing in html metadata

 hyperspy.misc.utils.closest_power_of_two (n)

 hyperspy.misc.utils.create_map_objects (function, nav_size, iterating_kwags, **kwags)
 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.\texttt{is\_hyperspy\_signal}(input\_object)
    Check if an object is a Hyperspy Signal
    \textbf{Parameters} input\_object (object) – Object to be tests
    \textbf{Returns} If true the object is a subclass of hyperspy.signal.BaseSignal
    \textbf{Return type} bool

hyperspy.misc.utils.\texttt{isiterable}(obj)

hyperspy.misc.utils.\texttt{iterable\_not\_string}(thing)

hyperspy.misc.utils.\texttt{map\_result\_construction}(signal, inplace, result, ragged, sig\_shape=None, lazy=False)

hyperspy.misc.utils.\texttt{multiply}(iterable)
    Return product of sequence of numbers.
    Equivalent of functools.reduce(operator.mul, iterable, 1).

    \begin{verbatim}
    >>> product([2**8, 2**30])
    274877906944
    >>> product([])
    1
    \end{verbatim}

hyperspy.misc.utils.\texttt{ordinal}(value)
    Converts zero or a positive integer (or their string representations) to an ordinal value.

    \begin{verbatim}
    >>> for i in range(1,13):
    ...     ordinal(i)
    ...
    '1st'
    '2nd'
    '3rd'
    '4th'
    '5th'
    '6th'
    '7th'
    '8th'
    '9th'
    '10th'
    '11th'
    '12th'
    >>> for i in (100, '111', '112',1011):
    ...     ordinal(i)
    ...
    '100th'
    '111th'
    '112th'
    '1011th'
    \end{verbatim}
Notes

Author: Serdar Tungoren http://code.activestate.com/recipes/576888-format-a-number-as-an-ordinal/ MIT license

```python
hyperspy.misc.utils.parse_quantity(quantity, opening='(', closing=')')
```

Parse quantity of the signal outputting quantity and units separately. It looks for the last matching opening and closing separator.

**Parameters**

- `quantity` *(string)*
- `opening` *(string)* – Separator used to define the beginning of the units
- `closing` *(string)* – Separator used to define the end of the units

**Returns**

- `quantity_name` *(string)*
- `quantity_units` *(string)*

```python
hyperspy.misc.utils.print_html(f_text, f_html)
```

Print html version when in Jupyter Notebook

```python
hyperspy.misc.utils.replace_html_symbols(str_value)
```

Escapes any &, < and > tags that would become invisible when printing html

```python
hyperspy.misc.utils.rollolem(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*

```python
hyperspy.misc.utils.shorten_name(name, req_l)
```

```python
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”.

```python
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`.  

3.1. hyperspy package
• **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*)

hyperspy.misc.utils.**str2num** (*string, **kargs*)

Transform a a table in string form into a numpy array

**Parameters**

- **string** (*string*) –

**Returns**

**Return type** numpy array

hyperspy.misc.utils.**strlist2enumeration** (*lst*)

hyperspy.misc.utils.**swapelem** (*obj, i, j*)

Swaps element having index *i* with element having index *j* in object *obj* IN PLACE.

**Example**

```python
>>> L = ['a', 'b', 'c']
>>> spwapelem(L, 1, 2)
>>> print(L)
['a', 'c', 'b']
```

hyperspy.misc.utils.**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)>,
 <BaseSignal, title: , dimensions: (2|2)>,
 <BaseSignal, title: , dimensions: (2, 2|2)>]
>>> hs.transpose(signal1, signal2, signal3, signal_axes=['Energy'])
```

```python
hyperspy.misc.utils.underline(line, character=':')
Return the line underlined.
```

Module contents

hyperspy.models package

Submodules

hyperspy.models.edsmodel module

```python
class hyperspy.models.edsmodel.EDSModel(spectrum,  
    auto_background=True,  
    auto_add_lines=True, *args, **kwargs)
Bases: hyperspy.models.model1d.Model1D
Build and fit a model of an EDS Signal1D.

Parameters

• spectrum (EDSSpectrum (or any EDSSpectrum subclass) instance.) –

• auto_add_lines (bool) – 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 (bool) – 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)
```

```python
property _active_background_components
property _active_xray_lines
```

3.1. hyperspy package 517
_make_position_adjuster (component, fix_it, show_label)

_set_energy_offset (xray_lines, ref)
Adjust the width of all lines and set the fitted energy resolution to the spectrum

Parameters

• xray_lines (list of str or 'all_alpha') – The X-ray lines. If ‘all_alpha’,
  fit all using all alpha lines

• ref (list of float) – The centres, before fitting, of the X-ray lines included

_set_energy_resolution (xray_lines, *args, **kwargs)
Adjust the width of all lines and set the fitted energy resolution to the spectrum

Parameters xray_lines (list of str or 'all_alpha') – The X-ray lines. If
‘all_alpha’, fit all using all alpha lines

_set_energy_scale (xray_lines, ref)
Adjust the width of all lines and set the fitted energy resolution to the spectrum

Parameters

• xray_lines (list of str or 'all_alpha') – The X-ray lines. If ‘all_alpha’,
  fit all using all alpha lines

• ref (list of float) – The centres, before fitting, of the X-ray lines included

_twin_xray_lines_offset (xray_lines)
Twin the offset of the peaks

Parameters xray_lines (list of str or 'all_alpha') – The X-ray lines. If
‘all_alpha’, fit all using all alpha lines

_twin_xray_lines_scale (xray_lines)
Twin the scale of the peaks

Parameters xray_lines (list of str or 'all_alpha') – The X-ray lines. If
‘all_alpha’, fit all using all alpha lines

_twin_xray_lines_width (xray_lines)
Twin the width of the peaks

The twinning models the energy resolution of the detector

Parameters xray_lines (list of str or 'all_alpha') – The X-ray lines. If
‘all_alpha’, fit all using all alpha lines

add_family_lines (xray_lines='from_elements')
Create the X-ray-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 dictionary including at least the following fields:

- components: a list of dictionaries of components, one per component
- _whitelist: a dictionary with keys used as references for saved attributes, for more information, see export_to_dictionary()
- any field from _whitelist.keys()

Return type
dict

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’, calibrate 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 multifit

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', 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', 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', 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, plot_result=False, only_one=True, only_lines='a', **kwargs)*
Return the fitted intensity of the X-ray lines.

Return the area under the gaussian corresping 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.edsmodel._get_offset(diff)
```

```
hyperspy.models.edsmodel._get_scale(E1, E_ref1, fact)
```

```
hyperspy.models.edsmodel._get_sigma(E, E_ref, units_factor, return_f=False)
```

Calculates an approximate sigma value, 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.models.edsmodel._get_weight(element, line, weight_line=None)`

**hyperspy.models.edssemmodel module**

```python
class hyperspy.models.edssemmodel.EDSSEMModel(spectrum, auto_background=True, auto_add_lines=True, *args, **kwargs)
```

Build and fit a model to EDS data acquired in the SEM.

Parameters

- `spectrum(EDSSEMSpectrum)`
- `auto_add_lines (bool)` – 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 (bool)` – 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**

```python
class hyperspy.models.edstemmodel.EDSTEMModel(spectrum, auto_background=True, auto_add_lines=True, *args, **kwargs)
```

Build and fit a model to EDS data acquired in the TEM.

Parameters

- `spectrum(EDSTEMSpectrum)`
- `auto_add_lines (bool)` – 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 (bool)` – 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)`
class hyperspy.models.eelsmodel.EELSModel

Bases: hyperspy.models.model1d.Model1D

Build an EELS model

Parameters

- spectrum (a Signal1D (or any Signal1D subclass) instance) –
- auto_background (bool) – 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 (bool) – 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()

property _active_background_components

property _active_edges

_add_edges_from_subshells_names (e_shells=None)

Create the Edge instances and configure them appropriately :param e_shells: :type e_shells: list of strings

_classify_components ()

Classify components between background and ionization edge components.

This method should be called everytime that components are added and removed. An ionization edge becomes background when its onset falls to the left of the first non-masked energy channel. The ionization edges are stored in a list in the edges attribute. They are sorted by increasing onset_energy. The background components are stored in _background_components.

_fit_edge (edgenumber, start_energy=None, **kwargs)

_get_first_ionization_edge_energy (start_energy=None)

Calculate the first ionization edge energy.

Returns iee – The first ionization edge energy or None if no edge is defined in the model.

Return type float or None

_get_start_energy (start_energy=None)

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.
```python
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 (kind='std', **kwargs)
Fits the model to the experimental data.

Read more in the User Guide.

Parameters

• kind ("std", "smart"), default "std") – If “std”, performs standard fit. If “smart”, performs a smart fit - for more details see the User Guide.

• optimizer (str or None, default None) – The optimization algorithm used to perform the fitting.
  - “lm” performs least-squares optimization using the Levenberg-Marquardt algorithm, and supports bounds on parameters.
- "trf" performs least-squares optimization using the Trust Region Reflective algorithm, and supports bounds on parameters.
- "dogbox" performs least-squares optimization using the dogleg algorithm with rectangular trust regions, and supports bounds on parameters.
- "odr" performs the optimization using the orthogonal distance regression (ODR) algorithm. It does not support bounds on parameters. See scipy.odr for more details.
- All of the available methods for scipy.optimize.minimize() can be used here. See the User Guide documentation for more details.
- "Differential Evolution" is a global optimization method. It does support bounds on parameters. See scipy.optimize.differential_evolution() for more details on available options.
- "Dual Annealing" is a global optimization method. It does support bounds on parameters. See scipy.optimize.dual_annealing() for more details on available options. Requires scipy >= 1.2.0.
- "SHGO" (simplicial homology global optimization" is a global optimization method. It does support bounds on parameters. See scipy.optimize.shgo() for more details on available options. Requires scipy >= 1.2.0.

  - **loss_function** ({"ls", "ML-poisson", "huber", callable}, default "ls") – The loss function to use for minimization. Only "ls" is available if optimizer is one of ["lm", "trf", "dogbox", "odr"].
    - "ls" minimizes the least-squares loss function.
    - "ML-poisson" minimizes the negative log-likelihood for Poisson-distributed data. Also known as Poisson maximum likelihood estimation (MLE).
    - "huber" minimize the Huber loss function. The delta value of the Huber function is controlled by the huber_delta keyword argument (the default value is 1.0).
    - callable supports passing your own minimization function.
  - **grad** ({"fd", "analytical", callable, None}, default "fd") – Whether to use information about the gradient of the loss function as part of the optimization. This parameter has no effect if optimizer is a derivative-free or global optimization method.
    - "fd" uses a finite difference scheme (if available) for numerical estimation of the gradient. The scheme can be further controlled with the fd_scheme keyword argument.
    - "analytical" uses the analytical gradient (if available) to speed up the optimization, since the gradient does not need to be estimated.
    - callable should be a function that returns the gradient vector.
    - None means that no gradient information is used or estimated. Not available if optimizer is in ["lm", "trf", "dogbox"].
  - **bounded** (bool, default False) – If True, performs bounded parameter optimization if supported by optimizer.
  - **update_plot** (bool, default False) – If True, the plot is updated during the optimization process. It slows down the optimization, but it enables visualization of the optimization progress.
print_info (bool, default False) – If True, print information about the fitting results, which are also stored in model.fit_output in the form of a scipy.optimize.OptimizeResult object.

return_info (bool, default True) – If True, returns the fitting results in the form of a scipy.optimize.OptimizeResult object.

fd_scheme (str {"2-point", "3-point", "cs"}, default "2-point") – If grad='fd', selects the finite difference scheme to use. See scipy.optimize.minimize() for details. Ignored if optimizer is "lm", "trf" or "dogbox".

**kwargs (keyword arguments) – Any extra keyword argument will be passed to the chosen optimizer. For more information, read the docstring of the optimizer of your choice in scipy.optimize.

Returns

Return type None

See also:

- fit()
- 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()`, `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** (\( preedge\_safe\_window\_width=2, il=0 \))
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 adjustament 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 EELS edges in a cascade style.
The fitting procedure acts in iterative manner along the energy-loss-axis. First it fits only the background up to the first edge. It continues by deactivating all edges except the first one, then performs the fit. Then it only activates the the first two, fits, and repeats this until all edges are fitted simultaneously.

Other, non-EELSCLEdge components, are never deactivated, and fitted on every iteration.

**Parameters**

- **start_energy** *(float, None)* – If float, limit the range of energies from the left to the given value.

- **optimizer** *(str or None, default None)* – The optimization algorithm used to perform the fitting.
  - ”lm” performs least-squares optimization using the Levenberg-Marquardt algorithm, and supports bounds on parameters.
  - ”trf” performs least-squares optimization using the Trust Region Reflective algorithm, and supports bounds on parameters.
  - ”dogbox” performs least-squares optimization using the dogleg algorithm with rectangular trust regions, and supports bounds on parameters.
  - ”odr” performs the optimization using the orthogonal distance regression (ODR) algorithm. It does not support bounds on parameters. See `scipy.odr` for more details.
  - All of the available methods for `scipy.optimize.minimize()` can be used here. See the User Guide documentation for more details.
  - ”Differential Evolution” is a global optimization method. It does support bounds on parameters. See `scipy.optimize.differential_evolution()` for more details on available options.
  - ”Dual Annealing” is a global optimization method. It does support bounds on parameters. See `scipy.optimize.dual_annealing()` for more details on available options. Requires `scipy >= 1.2.0`.
  - ”SHGO” (simplicial homology global optimization) is a global optimization method. It does support bounds on parameters. See `scipy.optimize.shgo()` for more details on available options. Requires `scipy >= 1.2.0`.

- **loss_function** *("ls", "ML-poisson", "huber", callable, default "ls")* – The loss function to use for minimization. Only ”ls” is available if optimizer is one of ["lm", "trf", "dogbox", "odr"].
  - ”ls” minimizes the least-squares loss function.
  - ”ML-poisson” minimizes the negative log-likelihood for Poisson-distributed data. Also known as Poisson maximum likelihood estimation (MLE).
  - ”huber” minimize the Huber loss function. The delta value of the Huber function is controlled by the `huber_delta` keyword argument (the default value is 1.0)
  - callable supports passing your own minimization function.

- **grad** *("fd", "analytical", callable, None, default "fd")* – Whether to use information about the gradient of the loss function as part of the optimization. This parameter has no effect if optimizer is a derivative-free or global optimization method.
  - ”fd” uses a finite difference scheme (if available) for numerical estimation of the gradient. The scheme can be further controlled with the `fd_scheme` keyword argument.
- "analytical" uses the analytical gradient (if available) to speed up the optimization, since the gradient does not need to be estimated.
- callable should be a function that returns the gradient vector.
- None means that no gradient information is used or estimated. Not available if optimizer is in ['lm', 'trf', 'dogbox'].

- **bounded** *(bool, default False)* – If True, performs bounded parameter optimization if supported by optimizer.
- **update_plot** *(bool, default False)* – If True, the plot is updated during the optimization process. It slows down the optimization, but it enables visualization of the optimization progress.
- **print_info** *(bool, default False)* – If True, print information about the fitting results, which are also stored in model.fit_output in the form of a scipy.optimize.OptimizeResult object.
- **return_info** *(bool, default True)* – If True, returns the fitting results in the form of a scipy.optimize.OptimizeResult object.
- **fd_scheme** *(str {"2-point", "3-point", "cs"}, default "2-point")* – If grad='fd', selects the finite difference scheme to use. See scipy.optimize.minimize() for details. Ignored if optimizer is "lm", "trf" or "dogbox".
- ****kwargs** *(keyword arguments)* – Any extra keyword argument will be passed to the chosen optimizer. For more information, read the docstring of the optimizer of your choice in scipy.optimize.

See also:
- `fit()`
- `multifit()`
- `fit()`

`suspend_auto_fine_structure_width()`
Disable the automatic adjustment 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

_fit_fired()

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_in_pixels(il=None, i2=None)

_add_signal_range_in_pixels 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)*

_check_analytical_jacobian()

_check_analytical_jacobian checks all components have analytical gradients.

If they do, return True and an empty string. If they do not, return False and an error message.

_close_plot()

Close_plot

_static_connect_component_line(component)

_static_connect_component_lines()

_disable_plot_component(component)

_static_disconnect_component_line(component)
_disconnect_component_lines()  
_errfunc (param, y, weights=None)  
_function4odr (param, x)  
_gradient_huber (param, y, weights=None, huber_delta=None)  
_gradient_ls (param, y, weights=None)  
_gradient_ml (param, y, weights=None)  
_hubers_loss_function (param, y, weights=None, huber_delta=None)  
_jacobian (param, y, weights=None)  
_jacobian4odr (param, x)  
_make_position_adjuster (component, fix_it, show_label)  
_model2plot (axes_manager, out_of_range2nans=True)  
_on_position_widget_close (widget)  
_on_widget_moved (widget)  
_plot_component (component)  
_poisson_likelihood_function (param, y, weights=None)  

Returns the likelihood function of the model for the given data and parameters

_remove_signal_range_in_pixels (i1=None, i2=None)

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)* –

_reverse_lookup_position_widget (widget)  
_set_signal_range_in_pixels (i1=None, i2=None)

Use only the selected spectral range in the fitting routine.

Parameters

- **i1** *(Int)* –
- **i2** *(Int)* –

Notes

To use the full energy range call the function without arguments.

static _update_component_line (component)  
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.

- `fix_them` *(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.

- `show_label` *(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. Note that ROIs can be used in place of a tuple.

- `estimate_parameters` *(bool, default True)* – If True will check if the component has an estimate_parameters function, and use it to estimate the parameters in the component.

- `fit_independent` *(bool, default False)* – If True, all other components are disabled. If False, all other component parameters are fixed.

- `display` *(bool)* – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.
• **toolkit** (*str, iterable of 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.

• **kwargs** (*dict*) — All extra keyword arguments are passed to the py:meth:`~hyperspy.model.BaseModel.fit` or py:meth:`~hyperspy.model.BaseModel.multifit` method, depending if `only_current` is True or False.

### Examples

Signal range set interactively

```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_2convolution_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`

**hyperspy.models.model2d module**

class `hyperspy.models.model2d.Model2D(signal2D, dictionary=None)`
Bases: `hyperspy.model.BaseModel`

Model and data fitting for two dimensional signals.

A model is constructed as a linear combination of `components2D` that are added to the model using `append()` or `extend()`. There are many predefined components available in the 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.

property `signal`
It contains the data to fit.

Type `Signal2D instance`

property `chisq`
Chi-squared of the signal (or np.nan if not yet fit)

Type `A Signal of floats`

property `dof`
Degrees of freedom of the signal (0 if not yet fit)

Type `A Signal of integers`

property `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
tools 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

_add_signal_range_in_pixels(i1=None, i2=None)
_check_analytical_jacobian()
   Check all components have analytical gradients.
   If they do, return True and an empty string. If they do not, return False and an error message.

static _connect_component_line (component)
_stderrfunc (param, y, weights=None)
_function4odr (param, x)
_gradient_huber (param, y, weights=None, huber_delta=None)
_gradient_ls (param, y, weights=None)
_gradient_ml (param, y, weights=None)
_huber_loss_function (param, y, weights=None, huber_delta=None)
_jacobian (param, y, weights=None)
_jacobian4odr (param, x)
_model2plot (axes_manager, out_of_range2nans=True)
_plot_component (component)
_poisson_likelihood_function (param, y, weights=None)
_remove_signal_range_in_pixels(i1=None, i2=None)
_set_signal_range_in_pixels(i1=None, i2=None)
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

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(tolerance)
    Bases: hyperSpy.samfire_utils.goodness_of_fit_tests.test_general.goodness_test
    map(model, mask)
    test(model, ind)

3.1. hyperspy package
hyperspy.samfire_utils.goodness_of_fit_tests.test_general module

class hyperspy.samfire_utils.goodness_of_fit_tests.test_general.goodness_test
    Bases: object
    _tolerance = None
    property tolerance

Module contents

hyperspy.samfire_utils.segmenters package

Submodules

hyperspy.samfire_utils.segmenters.histogram module

class hyperspy.samfire_utils.segmenters.histogram.HistogramSegmenter(bins='fd')
    Bases: object
    Histogram 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 frequences.
    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**

```python
class hyperspy.samfire_utils.global_strategies.HistogramStrategy (bins='fd')
    Bases: hyperspy.samfire_utils.strategy.GlobalStrategy
```

**hyperspy.samfire_utils.local_strategies module**

```python
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.
```

```python
hyperspy.samfire_utils.local_strategies.exp_decay (distances)
    Exponential decay function.
```

**hyperspy.samfire_utils.samfire_kernel module**

```python
hyperspy.samfire_utils.samfire_kernel.multi_kernel (ind, m_dic, values, optional_components, _args, result_q, test_dict)
hyperspy.samfire_utils.samfire_kernel.single_kernel (model, ind, values, optional_components, _args, test)
```

**hyperspy.samfire_utils.samfire_pool module**

```python
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 multiprocessing, or connects and sets up ipyparallel load_balanced_view.
    Ipyparallel is managed directly, but multiprocessing pool is managed via three of queues:
    • Shared by all (master and workers) for distributing “load-balanced” work.
    • Shared by all (master and workers) for sending results back to the master
    • Individual queues from master to each worker. For setting up and addressing individual workers in general. This one is checked with higher priority in workers.
    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
```

3.1. hyperspy package 543
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
property _not_too_long
    Returns bool if it has been too long after receiving the last result, probably meaning some of the workers
timed out or hung.

_timestep_set (value)

add_jobs (needed_number=None)
    Adds jobs to the job queue that is consumed by the workers.
    Parameters needed_number ((None, int)) – The number of jobs to add. If None (de-
default), adds need_pixels

collect_results (timeout=None)
    Collects and parses all results, currently not processed due to being in the queue.
    Parameters timeout ((None, float)) – the time to wait when collecting results. If None,
the default timeout is used

property need_pixels
    Returns the number of pixels that should be added to the processing queue. At most is equal to the number
of workers.

parse (value)
    Parses the value, returned from the workers.
    Parameters value (tuple of the form (keyword, the_rest)) – Keyword cur-
rently can be one of ['pong', 'Error', 'result']. For each of the keywords, “the_rest” is a tuple
of different elements, but generally the first one is always the worker_id that the result came
from. In particular:
    • ('pong', (worker_id, pid, pong_time, optional_message_str))
    • ('Error', (worker_id, error_message_string))
    • ('result', (worker_id, pixel_index, result_dict, bool_if_result_converged))

ping_workers (timeout=None)
    Pings the workers and records one-way trip time and (if available) pid of the worker.
    Parameters timeout ((None, float)) – the time to wait when collecting results after
sending out the ping. If None, the default timeout is used

prepare_workers (samfire)
    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_pool._walk_compute(athing)

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

    __array_views_to_copies()
    __collect_values()
    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)
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.

    __package_values__()
        Packages he current values to be sent to the segmenter

    __saved_values__ = None

    __update_database__(ind, count)
        Updates the database with current values

    __update_marker__(ind)
        Updates the SAMFire marker in the given pixel

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).

    __get_distance_array__(shape, ind)
        Calculates the array of distances (within radii) from the given pixel. Deals with borders well.

        Parameters
            • shape (tuple) – the shape of the original array
            • ind (tuple) – the index to calculate the distances from

        Returns
            • ans (numpy array) – the array of distances
            • slices (tuple of slices) – slices to slice the original marker to get the correct part of the array
            • centre (tuple) – the centre index in the sliced array
            • mask (boolean numpy array) – a binary mask for the values to consider

    __mask_all__ = None

    __radii__ = None

    __radii_changed__ = True
_samf = None
_untruncated = None
_update_database(*ind, count)
    Dummy method for compatibility
_update_marker(*ind)
    Updates the marker with the spatially decaying envelope around calculated pixels.
    Parameters
    *ind (tuple) – the index of the pixel to “spread” the envelope around.

_weight = None

_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: {parame-
    ter_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.utils package

Submodules

hyperspy.utils.eds module
hyperSpy Documentation, Release 1.6.0

hyperSpy.utils.markers 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

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

- 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.

- AICc(model)

- 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.

_setup_ipyparallel()  
_setup_multiprocessing()  
_timestep = 0  
_timestep_get()  
_timestep_set(value)
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.peakfinders2D module

hyperspy.utils.peakfinders2D._fast_mean (X)
    JIT-compiled mean of array.

    Parameters X (numpy.ndarray) – Input array.

    Returns mean – Mean of X.

    Return type float

Notes

Used by scipy.ndimage.generic_filter in the find_peaks_stat method to reduce overhead of repeated Python
function calls. See https://github.com/scipy/scipy/issues/8916 for more details.

hyperspy.utils.peakfinders2D._fast_std (X)
    JIT-compiled standard deviation of array.

    Parameters X (numpy.ndarray) – Input array.

    Returns std – Standard deviation of X.

    Return type float
Notes

Used by scipy.ndimage.generic_filter in the find_peaks_stat method to reduce overhead of repeated Python function calls. See https://github.com/scipy/scipy/issues/8916 for more details.

hyperspy.utils.peakfinders2D.clean_peaks(peaks)
Sort array of peaks and deal with no peaks being found.

Parameters

- **peaks** (*numpy.ndarray*) — Array of found peaks.

Returns

- **peaks** (*numpy.ndarray*) — Sorted array, first by peaks[:,1] (y-coordinate) then by peaks[:,0] (x-coordinate), of found peaks.
- **NO_PEAKS** (*str*) — Flag indicating no peaks found.

hyperspy.utils.peakfinders2D.find_local_max(z,**kwargs)
Method to locate positive peaks in an image by local maximum searching.

This function wraps skimage.feature.peak_local_max() function and sorts the results for consistency with other peak finding methods.

Parameters

- **z** (*numpy.ndarray*) — 2-d array of intensities
- **kwargs** — Additional parameters to be passed to the algorithm. See blob_dog documentation for details: http://scikit-image.org/docs/dev/api/skimage.feature.html#blob-dog

Returns

- **peaks** — (n_peaks, 2) Peak pixel coordinates.

Return type

- *numpy.ndarray*

hyperspy.utils.peakfinders2D.find_peaks_dog(z, min_sigma=1.0, max_sigma=50.0, sigma_ratio=1.6, threshold=0.2, overlap=0.5, exclude_border=False)
Method to locate peaks via the Difference of Gaussian Matrices method.

This function wraps skimage.feature.blob_dog() function and sorts the results for consistency with other peak finding methods.

Parameters

- **z** (*numpy.ndarray*) — 2-d array of intensities
- **max_sigma**, **sigma_ratio**, **threshold**, **overlap**, **exclude_border** — Additional parameters to be passed to the algorithm. See blob_dog documentation for details: http://scikit-image.org/docs/dev/api/skimage.feature.html#blob-dog

Returns

- **peaks** — (n_peaks, 2) Peak pixel coordinates.

Return type

- *numpy.ndarray*
Notes

While highly effective at finding even very faint peaks, this method is sensitive to fluctuations in intensity near the edges of the image.

```python
hyperspy.utils.peakfinders2D.find_peaks_log(z, min_sigma=1.0, max_sigma=50.0, num_sigma=10, threshold=0.2, overlap=0.5, log_scale=False, exclude_border=False)
```

Method to locate peaks via the Laplacian of Gaussian Matrices method.

This function wraps `skimage.feature.blob_log()` function and sorts the results for consistency with other peak finding methods.

**Parameters**

- `z` (*numpy.ndarray*) – Array of image intensities.
- `min_sigma`, `num_sigma`, `threshold`, `overlap`, `log_scale`, `exclude_border` (*min_sigma*,) – Additional parameters to be passed to the `blob_log` method of the `scikit-image` library. See its documentation for details: http://scikit-image.org/docs/dev/api/skimage.feature.html#blob-log

**Returns**

- `peaks` – (n_peaks, 2) Peak pixel coordinates.
- `Return type` *numpy.ndarray*

```python
hyperspy.utils.peakfinders2D.find_peaks_max(z, alpha=3.0, distance=10)
```

Method to locate positive peaks in an image by local maximum searching.

**Parameters**

- `alpha` (*float*) – Only maxima above `alpha * sigma` are found, where `sigma` is the standard deviation of the image.
- `distance` (*int*) – When a peak is found, all pixels in a square region of side `2 * distance` are set to zero so that no further peaks can be found in that region.

**Returns**

- `peaks` – (n_peaks, 2) Peak pixel coordinates.
- `Return type` *numpy.ndarray*

```python
hyperspy.utils.peakfinders2D.find_peaks_minmax(z, distance=5.0, threshold=10.0)
```

Method to locate the positive peaks in an image by comparing maximum and minimum filtered images.

**Parameters**

- `distance` (*float*) – Expected distance between peaks.
- `threshold` (*float*) – Minimum difference between maximum and minimum filtered images.

**Returns**

- `peaks` – (n_peaks, 2) Peak pixel coordinates.
- `Return type` *numpy.ndarray*

```python
hyperspy.utils.peakfinders2D.find_peaks_stat(z, alpha=1.0, window_radius=10, convergence_ratio=0.05)
```

Method to locate positive peaks in an image based on statistical refinement and difference with respect to mean intensity.

**Parameters**

- `window_radius` (*float*) – Expected distance between peaks.
- `convergence_ratio` (*float*) – Minimum difference between maximum and minimum filtered images.

**Returns**

- `peaks` – (n_peaks, 2) Peak pixel coordinates.
- `Return type` *numpy.ndarray*
• \texttt{z} (\texttt{numpy.ndarray}) -- Array of image intensities.

• \texttt{alpha} (float) -- Only maxima above \texttt{alpha * sigma} are found, where \texttt{sigma} is the local, rolling standard deviation of the image.

• \texttt{window\_radius} (int) -- The pixel radius of the circular window for the calculation of the rolling mean and standard deviation.

• \texttt{convergence\_ratio} (float) -- The algorithm will stop finding peaks when the proportion of new peaks being found is less than \texttt{convergence\_ratio}.

\textbf{Returns peaks} -- \texttt{(n\_peaks, 2)} Peak pixel coordinates.

\textbf{Return type} \texttt{numpy.ndarray}

\textbf{Notes}

Implemented as described in the PhD thesis of Thomas White, University of Cambridge, 2009, with minor modifications to resolve ambiguities.

The algorithm is as follows:

1. Adjust the contrast and intensity bias of the image so that all pixels have values between 0 and 1.
2. For each pixel, determine the mean and standard deviation of all pixels inside a circle of radius 10 pixels centered on that pixel.
3. If the value of the pixel is greater than the mean of the pixels in the circle by more than one standard deviation, set that pixel to have an intensity of 1. Otherwise, set the intensity to 0.
4. Smooth the image by convolving it twice with a flat 3x3 kernel.
5. Let \( k = \frac{1}{2} - \mu \)/\texttt{sigma} where \( \mu \) and \texttt{sigma} are the mean and standard deviations of all the pixel intensities in the image.
6. For each pixel in the image, if the value of the pixel is greater than \( \mu + k \texttt{sigma} \) set that pixel to have an intensity of 1. Otherwise, set the intensity to 0.
7. Detect peaks in the image by locating the centers of gravity of regions of adjacent pixels with a value of 1.
8. Repeat #4-7 until the number of peaks found in the previous step converges to within the user defined \texttt{convergence\_ratio}.

\texttt{hyperspy.utils.peakfinders2D.find\_peaks\_xc(z, template, distance=5, threshold=0.5, **kwargs)}

Find peaks in the cross correlation between the image and a template by using the \texttt{find\_peaks\_minmax()} function to find the peaks on the cross correlation result obtained using the \texttt{skimage.feature.match\_template()} function.

\textbf{Parameters}

• \texttt{z} (\texttt{numpy.ndarray}) -- Array of image intensities.

• \texttt{template} (\texttt{numpy.ndarray (square)}) -- Array containing a single bright disc, similar to those to detect.

• \texttt{distance} (float) -- Expected distance between peaks.

• \texttt{threshold} (float) -- Minimum difference between maximum and minimum filtered images.

• **\texttt{kwargs} (dict) -- Keyword arguments to be passed to the \texttt{skimage.feature.match\_template()} function.
Returns (n_peaks, 2) Array of peak coordinates.

Return type numpy.ndarray

hyperspy.utils.peakfinders2D.find_peaks_zaefferer(z, grad_threshold=0.1, window_size=40, distance_cutoff=50.0)

Method to locate positive peaks in an image based on gradient thresholding and subsequent refinement within masked regions.

Parameters

- z (ndarray) – Matrix of image intensities.
- grad_threshold (float) – The minimum gradient required to begin a peak search.
- window_size (int) – The size of the square window within which a peak search is conducted. If odd, will round down to even. The size must be larger than 2.
- distance_cutoff (float) – The maximum distance a peak may be from the initial high-gradient point.

Returns peaks – (n_peaks, 2) Peak pixel coordinates.

Return type numpy.ndarray

Notes


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_types*

This includes *signal_types* from all installed packages that extend HyperSpy.

### Examples

```python
>>> hs.print_known_signal_types()
+---------------------+---------------------+--------------------+----------+
| signal_type         | aliases             | class name         | package  |
|---------------------+---------------------+--------------------+----------+
| dielectric function | dielectric function | DielectricFunction | hyperspy |
| EDS_SEM             | EDS_SEMSpectrum     | EDS_SEMSpectrum    | hyperspy |
| EDS_TEM             | EDS_TEM             | EDS_TEM            | hyperspy |
| TEM EELS            | EELS                | EELSSpectrum       | hyperspy |
| hologram            | HologramImage       | HologramImage      | hyperspy |
| MySignal            | 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 execting its gui method i.e. preferences.gui().
 - stack  Stack several signals.
 - interactive  Define operations that are automatically recomputed on event changes.
 - set_log_level  Convenience 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 executing its `gui` method i.e. `preferences.gui()`.
- `stack` Stack several signals.
- `interactive` Define operations that are automatically recomputed on event changes.
- `set_log_level` Convenience 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.

```python
hyperspy.api_nogui.get_configuration_directory_path()
```
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. \([nX, nY, \ldots, sX, sY, \ldots]\) 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)
<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>&lt;undefined&gt;</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>&lt;undefined&gt;</td>
</tr>
<tr>
<td>&lt;undefined&gt;</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>&lt;undefined&gt;</td>
</tr>
<tr>
<td>&lt;undefined&gt;</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>&lt;undefined&gt;</td>
</tr>
<tr>
<td>&lt;undefined&gt;</td>
<td>5</td>
<td></td>
<td>0</td>
<td>1</td>
<td>&lt;undefined&gt;</td>
</tr>
</tbody>
</table>

>>> s.axes_manager[0]
<Unnamed 0th axis, size: 4, index: 0>
>>> s.axes_manager[3]
<Unnamed 2nd axis, size: 2, index: 0>
>>> s.axes_manager[1]
<Unnamed 0th axis, size: 4, index: 0>
>>> s.axes_manager[2]
<Unnamed 3rd axis, size: 5>
>>> s.axes_manager[1].name = "y"
>>> s.axes_manager["y"]
<y axis, size: 3, index: 0>
>>> for i in s.axes_manager:
```
...     print(i, s.axes_manager.indices)
...
(0, 0, 0) (0, 0, 0)
(1, 0, 0) (1, 0, 0)
(2, 0, 0) (2, 0, 0)
(3, 0, 0) (3, 0, 0)
(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)

_am_indices_generator()

_append_axis(*args, **kwargs)

_array_indices_generator()

_axes_getter(y)

_convert_axes_to_same_units(axes, units, factor=0.25)

_get_axes_dicts(axes=None)

_get_axes_in_natural_order()

_get_axis_attribute_values(attr)

_get_data_slice(fill=None)

    Return a tuple of slice objects to slice the data.

    Parameters fill (None or iterable of (int, slice)) – If not None, fill the tuple
    of index int with the given slice.

_get_dimension_str()

_get_navigation_axes_dicts()

_get_positive_index(axis)

_get_signal_axes_dicts()

property _navigation_shape_in_array

_on_index_changed()

_on_offset_changed()
_on_scale_changed()

_on_size_changed()

_on_slice_changed()

_remove_one_axis(axis)
Remove the given Axis.

Raises ValueError – If the Axis is not present.

_repr_html_()

_set_axis_attribute_values(attr, values)
Set the given attribute of all the axes to the given value(s)

Parameters

• attr (string) – The DataAxis attribute to set.

• values (any) – If iterable, it must have the same number of items as axes are in this AxesManager instance. If not iterable, the attribute of all the axes are set to the given value.

property _signal_shape_in_array

_sort_axes()
Sort axes to align them.

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]. This method sort the axes in this way. Warning: this doesn’t sort the data axes.

_update_attributes()

_update_max_index()

_update_trait_handlers (remove=False)

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

_get_array_slices (slice_)
Returns a slice to slice the corresponding data axis without changing the offset and scale of the DataAxis.

Parameters slice (float, int, slice)

Returns my_slice

Return type slice

_get_index (value)

_get_name ()

_get_positive_index (index)

_index_changed (name, old, new)

_parse_string_for_slice (value)

_slice_me (slice_)
Returns a slice to slice the corresponding data axis and change the offset and scale of the DataAxis accordingly.

Parameters slice (float, int, slice)

Returns my_slice

Return type slice

_update_slice (value)
_value_changed(name, old, new)
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(v1, v2)
   Convert the given range to index range.
   When an out of the axis limits, the endpoint is used instead.
   Parameters v2 (v1,) – The end points of the interval in the axis units. v2 must be greater than v1.
class hyperspy.axes.UnitConversion(units=<undefined>, scale=1.0, offset=0.0)

Bases: object

_convert_compact_units(factor=0.25, inplace=True)

Convert units to “human-readable” units, which means with a convenient prefix.

Parameters

• factor(float (default: 0.25)) –

• is an adjustable value used to determine the prefix of ('factor') –

• units. The product factor * scale * size is passed to the
  (the) –

• to_compact method to determine the prefix. (pint) –

_convert_units(converted_units, inplace=True)

_get_index_from_value_with_units(value)

_get_quantity(attribute='scale')

_ignore_conversion(units)

_set_quantity(value, attribute='scale')

convert_to_units(units=None, inplace=True, factor=0.25)

Convert the scale and the units of the current axis. If the unit of measure is not supported by the pint library, the scale and units are not modified.

Parameters

• units({str | None}) – Default = None If str, the axis will be converted to the provided units. If “auto”, automatically determine the optimal units to avoid using too large or too small numbers. This can be tweaked by the factor argument.

• inplace(bool) – If True, convert the axis in place. if False return the scale, offset and units.

• factor(float (default: 0.25)) – ‘factor’ is an adjustable value used to determine the prefix of the units. The product factor * scale * size is passed to the pint to_compact method to determine the prefix.

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)
Bases: numpy.ndindex

hyperspy.axes.serpentine_iter(shape)
Similar to np.ndindex, but yields indices in serpentine pattern, like snake game
Code by Stackoverflow user Paul Panzer, from https://stackoverflow.com/questions/57366966/

hyperspy.component module

class hyperspy.component.Component (parameter_name_list)
Bases: traits.has_traits.HasTraits

_Component__axes_manager = None
_active = True
_active_is_multidimensional = False

property _axes_manager
_Component2plot (axes_manager, out_of_range2nans=True)

_component2plot
_create_active_array()

_create_arrays()

_estimate_parameters(signal)

_get_active()

_get_long_description()

_get_name()

_get_short_description()

property _is_navigation_multidimensional

_load_dictionary (dic)
Load data from dictionary.

Parameters
dict (dict) – A dictionary containing at least the following fields:

- _id_name: _id_name of the original parameter, used to create the dictionary. Has to match
  with the self._id_name

- parameters: a list of dictionaries, one per parameter of the component (see
  as_dictionary() documentation for more details)

- _whitelist: a dictionary, which keys are used as keywords to match with the parameter
  attributes. For more information see load_from_dictionary()

- any field from _whitelist.keys()

Returns
twin_dict – Dictionary of ‘id’ values from input dictionary as keys with all of the
parameters of the component, to be later used for setting up correct twins.

Return type
dict

_name = ''

_set_active (arg)

_set_name (value)

_store_active_value_in_array (value)
_update_free_parameters()

property active_is_multidimensional

as_dictionary (fullcopy=True)

Returns component as a dictionary. For more information on method and conventions, see py:meth:~hyperspy.misc.export_dictionary.export_to_dictionary

Parameters fullcopy (Bool (optional, False)) – Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

Returns
dic – A dictionary, containing at least the following fields:

• parameters: a list of dictionaries of the parameters, one per component.
• __whitelist: a dictionary with keys used as references saved attributes, for more information, see export_to_dictionary()
• any field from __whitelist.keys()

Return type dict

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 the value of the parameters of the model

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. :param only_free: If True, only free parameters will be printed. :type only_free: bool :param fancy: If True, attempts to print using html rather than text in the notebook. :type fancy: bool

**set_parameters_free** *(parameter_name_list=None)*

Sets parameters in a component to free.

Parameters **parameter_name_list** *(None or list of str, optional)* – If None, will set all the parameters to free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to free.

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 str, optional)* – If None, will set all the parameters to not free. If list of strings, will set all the parameters with the same name as the strings in parameter_name_list to not free.

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** *

### 3.1. hyperspy package

This is the only method normally called directly by client code. It defines the trait. The default implementation accepts an optional, unvalidated default value, and caller-supplied trait metadata.
Override this method whenever a different method signature or a validated default value is needed.

```python
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

    ext_force_positive
        If True, the parameter value is set to be the absolute value of the input value i.e. if we set Parameter.value
= -3, the value stored is 3 instead. This is useful to bound a value to be positive in an optimization without
actually using an optimizer that supports bounding.

        Type bool
**ext_bounded**

Similar to `ext_force_positive`, but in this case the bounds are defined by `bmin` and `bmax`. It is a better idea to use an optimizer that supports bounding though.

**Type**  
`bool`

**as_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.

_**Parameter**__ext_bounded = False

_**Parameter**__ext_force_positive = False

_**Parameter**__free = True

_**Parameter**__number_of_elements = 1

_**Parameter**__twin = None

_**Parameter**__value = 0

_**axes_manager** = None

_**bounds** = (None, None)

_**create_array** (*)

Create the map array to store the information in multidimensional datasets.

_**get_bmax** (*)

_**get_bmin** (*)

_**get_free** (*)

_**get_twin** (*)

_**get_value** (*)

_**load_dictionary** *(dictionary)*

Load data from dictionary.

**Parameters**  
`dict (dict)` A dictionary containing at least the following fields:

- _id_name: _id_name of the original parameter, used to create the dictionary. Has to match with the self._id_name.

- _whitelist: a dictionary, which keys are used as keywords to match with the parameter attributes. For more information see `load_from_dictionary()`

- any field from _whitelist.keys().

Returns **id_value** – the ID value of the original parameter, to be later used for setting up the correct twins

**Return type**  
`int`

**property _number_of_elements**

_**on_twin_update** *(value, twin=None)*

3.1. hyperspy package 571
HyperSpy Documentation, Release 1.6.0

```python
_set_bmax(arg)
_set_bmin(arg)
_set_free(arg)
_set_twin(arg)
_set_value(value)
_twin_function_expr = ''
_twin_inverse_function = None
_twin_inverse_function_expr = ''
_twin_inverse_sympy = None

as_dictionary (fullcopy=True)
Returns parameter as a dictionary, saving all attributes from self._whitelist.keys() For more information see py:meth:~hyperspy.misc.export_dictionary.export_to_dictionary

Parameters
fullcopy (Bool (optional, False)) – Copies of objects are stored, not references. If any found, functions will be pickled and signals converted to dictionaries

Returns
- _id_name: _id_name of the original parameter, used to create the dictionary. Has to match with the self._id_name
- _twins: a list of ids of the twins of the parameter
- _whitelist: a dictionary, which keys are used as keywords to match with the parameter attributes. For more information see export_to_dictionary()
- any field from _whitelist.keys()

Return type
A dictionary, containing at least the following fields

as_signal (field='values')
Get a parameter map as a signal object.

Please note that this method only works when the navigation dimension is greater than 0.

Parameters
field ({'values', 'std', 'is_set'}) – Field to return as signal.

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 *hs.py*.

**property ext_bounded**

**property ext_force_positive**

**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**

```python
>>> parameter.plot()
```

Set the minimum and maximum displayed values

```python
>>> parameter.plot(vmin=0, vmax=1)
```

**store_current_value_in_array()**

Store the value and std attributes.

See also:

*fetch()*, *assign_current_value_to_all()*

**property twin**

*twin_function = None*

**property twin_function_expr**

**property twin_inverse_function**

**property twin_inverse_function_expr**
 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_cl_edge.

Writing a new template is easy: see the user guide documentation on creating components.

For more details see each component docstring.

Arctan This is the legacy Arctan component dedicated.. Bleasdale Bleasdale function component. Doniach Doniach Sunjic lineshape. DoublePowerLaw Double power law component for EELS spectra. EELSArctan Arctan function component for EELS (with mi.. EELSCLEdge EELS core loss ionisation edge from hydrogen. 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-.. PESCoreLineShape.. PESVoigt Voigt component for photoemission spectros.. 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. SplitVoigt Split pseudo-Voigt.. Vignetting.. Voigt This is the legacy Voigt profile component.. VolumePlasmonDrude Drude volume plasmon energy loss function c..

 hyperspy.components2d module

Components that can be used to define a 2D model for e.g. 2D model fitting.

Writing a new template is easy: see the user guide documentation on creating components.

For more details see each component docstring.

Expression Create a component from a string expression. Gaussian2D Normalized 2D elliptical Gaussian function..

 hyperspy.conftest module

 hyperspy.decorators module

 hyperspy.decorators.interactive_range_selector(f)
 hyperspy.decorators.lazify(func, **kwargs)
 hyperspy.decorators.lazifyTestClass(*args, **kwargs)
 hyperspy.decorators.simple_decorator(decorator)

This decorator can be used to turn simple functions into well-behaved decorators, so long as the decorators are fairly simple. If a decorator expects a function and returns a function (no descriptors), and if it doesn’t modify function attributes or docstring, then it is eligible to use this. Simply apply @simple_decorator to your decorator and it will automatically preserve the docstring and function attributes of functions to which it is applied.

This decorator was taken from: http://wiki.python.org/moin/PythonDecoratorLibrary
hyperspy.defaults_parser module

```python
class hyperspy.defaults_parser.EDSConfig
    Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.EELSConfig
    Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.GUIs
    Bases: traits.has_traits.HasTraits

class hyperspy.defaults_parser.GeneralConfig
    Bases: traits.has_traits.HasTraits

    _logger_on_changed(old, new)

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

```python
class hyperspy.events.Event(doc=", arguments=None)
    Bases: object

    Create an Event object.

    Parameters
    • doc (str) – Optional docstring for the new Event.
    • arguments (iterable) – Pass to define the arguments of the trigger() function. Each
        element must either be an argument name, or a tuple containing the argument name
        and the argument’s default value.
```
Example

```python
>>> from hyperspy.events import Event
>>> Event()
<hyperspy.events.Event: set()>
>>> Event(doc="This event has a docstring!").__doc__
'This event has a docstring!'
>>> e1 = Event()
>>> e2 = Event(arguments=('arg1', ('arg2', None)))
>>> e1.trigger(arg1=12, arg2=43, arg3='str', arg4=4.3)  # Can trigger with whatever
>>> e2.trigger(arg1=11, arg2=22, arg3=3.4)
Traceback (most recent call last):
...  
TypeError: trigger() got an unexpected keyword argument 'arg3'
```

```python
_re_arg_name = re.compile('[a-zA-Z_][a-zA-Z0-9-_]*')

_trigger_maker (arguments)
   Dynamically creates a function with a signature equal to arguments.
   Ensures that trigger can only be called with the correct arguments

property arguments
   connect (function, kwarg='all')
      Connects a function to the event.

      Parameters
      • function (callable) – The function to call when the event triggers.
      • kwarg (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.

      Parameters
      • function (function) –
      • return_connection_kwargs (bool, default False) – If True, returns the kwarg 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.

**Example**

```python
>>> 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.

**Example**

```python
>>> 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(**kwargs**)**

Triggers the event. If the event is suppressed, this does nothing. Otherwise it calls all the connected functions with the arguments as specified when connected.

See also:

`suppress()`, `suppress_callback()`, `Events.suppress()`

**class hyperspy.events.EventSuppressor(*to_suppress)**

**Bases:** `object`

Object to enforce a variety of suppression types simultaneously

Targets to be suppressed can be added by the function `add()`, or given in the constructor. Valid targets are:

- **Event**: The entire Event will be suppressed
- **Events**: All events in th container will be suppressed
- (Event, callback): The callback will be suppressed in Event
- (Events, callback): The callback will be suppressed in each event in Events where it is connected.
- Any iterable collection of the above target types
Example

```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()
```

_functions_

```python
_add_single(target)
_is_target(candidate)
_is_tuple_target(candidate)
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

```python
suppress()
```

Use this function with a ‘with’ statement to temporarily suppress all events added. When the ‘with’ lock completes, the old suppression values will be restored.

See also:

- `Events.suppress()`, `Event.suppress()`, `Event.suppress_callback()`

```python
class hyperspy.events.Events
```

Events container.

All available events are attributes of this class.

```python
_update_doc()
```

Updates the doc to reflect the events that are contained

```python
suppress()
```

Use this function with a ‘with’ statement to temporarily suppress all callbacks of all events in the container. When the ‘with’ lock completes, the old suppression values will be restored.
Example

```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.WriteByteOrderError(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.MountainsMapFileError(msg='Corrupt Mountainsmap file')`  
  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

---

**3.1. hyperspy package**  

579
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 ({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 – Arguments to be passed to f.

    • **kwargs – Keyword arguments to be passed to f.

    recompute_out()

    update()

hyperspy.interactive._connect_events(event, to_connect)

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_chaged` event is selected. Otherwise the `Signal data_changed` event is selected. If None, `recompute_out` is not connected to any event. The default is “auto”. It is also possible to pass an iterable of events, in which case all the events are connected.

- ***args** – Arguments to be passed to f.

- ****kwargs** – Keyword arguments to be passed to f.

**hyperspy.io module**

**hyperspy.io._escape_square_brackets** *(text)*

Escapes pairs of square brackets in strings for `glob.glob()`.

**Parameters**
- **text** *(str)* – The text to escape

**Returns**
- The escaped string

**Return type** *(str)*

**Example**

```python
>>> # Say there are two files like this:
>>> # /home/data/afile[1x1].txt
>>> # /home/data/afile[1x2].txt
>>> path = "/home/data/afile[*].txt"
>>> glob.glob(path)
[]
>>> glob.glob(_escape_square_brackets(path))
['/home/data/afile[1x2].txt', '/home/data/afile[1x1].txt']
```

**hyperspy.io.assign_signal_subclass** *(dtype, signal_dimension, signal_type='', lazy=False)*

Given dtype, signal_dimension and signal_type, return the matching Signal subclass.

See `hs.print_known_signal_types()` for a list of known signal_types, and the developer guide for details on how to add new signal_types.

**Parameters**

- **dtype** *(dtype)* – Signal dtype

- **signal_dimension** *(int)* – Signal dimension
**signal_type** *(str, default "")* – Signal type. Optional. Will log a warning if it is unknown to HyperSpy.

**lazy** *(bool, default False)* – If True, returns the matching LazySignal subclass.

Returns

Return type  Signal or subclass

```python
hyperspy.io.dict2signal(signal_dict, lazy=False)
```

Create a signal (or subclass) instance defined by a dictionary

Parameters

**signal_dict** *(dictionary)* –

Returns

Return type  Signal or subclass

```python
hyperspy.io.load(filenames=None, signal_type=None, stack=False, stack_axis=None, new_axis_name='stack_element', lazy=False, convert_units=False, escape_square_brackets=False, **kwds)
```

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.

Depending on the number of datasets to load in the file, this function will return a HyperSpy signal instance or list of HyperSpy signal instances.

Any extra keyword is passed to the corresponding reader. For available options see their individual documentation.

Parameters

- **filenames** *(None, str or list of strings)* – 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 below). 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** *(None, "EELS", "EDS_SEM", "EDS_TEM", ",", str)* – 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** *(bool)* – 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** *(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** *(None, bool)* – 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** *(bool)* – If True, convert the units using the `convert_to_units` method of the `axes_manager`. If False, does nothing. The default is False.

- **escape_square_brackets** *(bool, default False)* – If True, and filenames is a str containing square brackets, then square brackets are escaped before wildcard matching with `glob.glob()`. If False, square brackets are used to represent character classes (e.g. `[a-z]` matches lowercase letters.

- **print_info** *(bool)* – For SEMPER un- and EMD (Berkeley)-files, if True (default is False) additional information read during loading is printed for a quick overview.

- **downsample** *(int (1-4095))* – For Bruker bcf files, if set to integer (>=2) (default 1) bcf is parsed into down-sampled size array by given integer factor, multiple values from original bcf pixels are summed forming downsampled pixel. This allows to improve signal and conserve the memory with the cost of lower resolution.

- **cutoff_at_kV** *(None, int, float)* – For Bruker bcf files, if set to numerical (default is None) bcf is parsed into array with depth cutoff at coresponding given energy. 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_spectrums 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_path** *(None, str or list of str, optional)* – For filetypes which support several datasets in the same file, this will only load the specified dataset.
Several datasets can be loaded by using a list of strings. Only for EMD (NCEM) and hdf5 (USID) files.

- **stack_group** *(bool, optional)* – Only for EMD NCEM. Stack datasets of groups with common name. Relevant for emd file version >= 0.5 where groups can be named ‘group0000’, ‘group0001’, etc.

- **ignore_non_linear_dims** *(bool, default is True)* – Only for HDF5 USID. If True, parameters that were varied non-linearly in the desired dataset will result in Exceptions. Else, all such non-linearly varied parameters will be treated as linearly varied parameters and a Signal object will be generated.

- **only_valid_data** *(bool, optional)* – Only for FEI emi/ser 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:

```python
>>> d = hs.load('file*.dm3')
```

Loading multiple files containing square brackets:

```python
>>> d = hs.load('file[*].dm3', escape_square_brackets=True)
```

Loading (potentially larger than the available memory) files lazily and stacking:

```python
>>> 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)*

Save hyperspy signal to a file.

A list of plugins supporting file saving can be found here: [http://hyperspy.org/hyperspy-doc/current/user_guide/io.html#supported-formats](http://hyperspy.org/hyperspy-doc/current/user_guide/io.html#supported-formats)

Any extra keyword is passed to the corresponding save method in the io_plugin. For available options see their individual documentation.
Parameters

- **filename** (*None or str*) – The filename to save the signal to.
- **signal** (*Hyperspy signal*) – The signal to be saved to file
- **overwrite** (*None or Bool (default, None]*) – If None and a file exists the user will be prompted to on whether to overwrite. If False and a file exists the file will not be written. If True and a file exists the file will be overwritten without prompting

**hyperspy.logger module**

`hyperspy.logger.set_log_level(level)`

Convenience function to set the log level of all hyperspy modules.

Note: The log level of all other modules are left untouched.

**Parameters level** (*int or str*) – The log level to set. Any values that `logging.Logger.setLevel()` accepts are valid. The default options are:

- 'CRITICAL'
- 'ERROR'
- 'WARNING'
- 'INFO'
- 'DEBUG'
- 'NOTSET'

**Example**

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

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 the append() or extend(). There are many predefined components available in the 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`

**_as_signal_iter (component_list=None, show_progressbar=None, data=None)**

**_bounds_as_tuple ()**  
Converts parameter bounds to tuples for least_squares()

**_calculate_chisq ()**

**_calculate_parameter_std (pcov, cost, ysize)**

**_close_plot ()**

**static _connect_component_line (component)**

**_connect_component_lines ()**
_connect_parameters2update_plot (components)

(converted_variance_to_weights ()

_disable_extBounding (components=None)

disable_plot_component (component)

static _disconnect_component_line (component)

disconnect_component_lines ()

disconnect_parameters2update_plot (components)

_enable_extBounding (components=None)

Errfunc4mpfit (p, fjac=None, x=None, y=None, weights=None)

Errfunc_sq (param, y, weights=None)

_fetch_values_from_p0 (p_std=None)

Fetch the parameter values from the output of the optimizer self.p0

Parameters p_std (array, optional) – array containing the corresponding standard deviation.

_get_component (thing)

_get_variance (only_current=True)

Return the variance taking into account the channel_switches. If only_current=True, the variance for the current navigation indices is returned, otherwise the variance for all navigation indices is returned.

_load_dictionary (dic)

Load data from dictionary.

Parameters dic (dict) – A dictionary containing at least the following fields:

• _whitelist: a dictionary with keys used as references of save attributes, for more information, see load_from_dictionary()

• components: a dictionary, with information about components of the model (see as_dictionary() documentation for more details)

• any field from _whitelist.keys()

_model2plot (axes_manager, out_of_range2nans=True)

_model_function (param)

property _plot_active

_set_boundaries (bounded=True)

Generate the boundary list.

Necessary before fitting with a boundary aware optimizer.

Parameters bounded (bool, default True) – If True, loops through the model components and populates the free parameter boundaries.

Returns

Return type None

_set_current_degrees_of_freedom ()

_set_mpfit_parameters_info (bounded=True)

Generate the boundary list for mpfit.
Parameters `bounded` *(bool, default True)* – If True, loops through the model components and populates the free parameter boundaries.

Returns

Return type None

```python
_set_p0()
```

```python
static _update_component_line(component)
```

```python
append(thing)
```

Add component to Model.

Parameters `thing` *(Component instance.) –*

```python
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

```python
dictionary – A dictionary including at least the following fields:
```

- components: a list of dictionaries of components, one per component
- _whitelist: a dictionary with keys used as references for saved attributes, for more information, see `export_to_dictionary()`
- any field from _whitelist.keys()

Return type dict

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)
```

```python
as_signal(component_list=None, out_of_range_to_nan=True, show_progressbar=None, out=None, parallel=None, max_workers=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 or bool)* – If True, perform computation in parallel using multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the *max_workers* argument.

• **max_workers** *(None or int)* – Maximum number of threads used when parallel=True. If None, defaults to min(32, os.cpu_count()).

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)
>>> m.append(l2)
>>> s1 = m.as_signal()
>>> s2 = m.as_signal(component_list=[l1])
```

assign_current_values_to_all *(components_list=None, mask=None)*

Set parameter values for all positions to the current ones.

**Parameters**

• **component_list** *(list of 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=None, 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=None, format='hspy', save_std=False, only_free=True, only_active=True)*

Export the results of the parameters of the model to the desired folder.
Parameters

- **folder** *(str or None)* – The path to the folder where the file will be saved. If None the current folder is used by default.

- **format** *(str)* – The extension of the file format. It must be one of the file formats 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 Component and 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** *(optimizer='lm', loss_function='ls', grad='fd', bounded=False, update_plot=False, print_info=False, return_info=True, fd_scheme='2-point', **kwargs)*

Fits the model to the experimental data.

Read more in the User Guide.

Parameters

- **optimizer** *(str or None, default None)* – The optimization algorithm used to perform the fitting.

  - “lm” performs least-squares optimization using the Levenberg-Marquardt algorithm, and supports bounds on parameters.

  - “trf” performs least-squares optimization using the Trust Region Reflective algorithm, and supports bounds on parameters.

  - “dogbox” performs least-squares optimization using the dogleg algorithm with rectangular trust regions, and supports bounds on parameters.

  - “odr” performs the optimization using the orthogonal distance regression (ODR) algorithm. It does not support bounds on parameters. See scipy.odr for more details.
All of the available methods for `scipy.optimize.minimize()` can be used here. See the User Guide documentation for more details.

“Differential Evolution” is a global optimization method. It does support bounds on parameters. See `scipy.optimize.differential_evolution()` for more details on available options.

“Dual Annealing” is a global optimization method. It does support bounds on parameters. See `scipy.optimize.dual_annealing()` for more details on available options. Requires scipy >= 1.2.0.

“SHGO” (simplicial homology global optimization) is a global optimization method. It does support bounds on parameters. See `scipy.optimize.shgo()` for more details on available options. Requires scipy >= 1.2.0.

- **loss_function**
  (("ls", "ML-poisson", "huber", callable),
default "ls") – The loss function to use for minimization. Only "ls" is available if `optimizer` is one of ["lm", "trf", "dogbox", "odr"].
  - "ls" minimizes the least-squares loss function.
  - "ML-poisson" minimizes the negative log-likelihood for Poisson-distributed data. Also known as Poisson maximum likelihood estimation (MLE).
  - "huber" minimize the Huber loss function. The delta value of the Huber function is controlled by the `huber_delta` keyword argument (the default value is 1.0).
  - callable supports passing your own minimization function.

- **grad**
  ("fd", "analytical", callable, None), default "fd"
  Whether to use information about the gradient of the loss function as part of the optimization. This parameter has no effect if optimizer is a derivative-free or global optimization method.
  - "fd" uses a finite difference scheme (if available) for numerical estimation of the gradient. The scheme can be further controlled with the `fd_scheme` keyword argument.
  - "analytical" uses the analytical gradient (if available) to speed up the optimization, since the gradient does not need to be estimated.
  - callable should be a function that returns the gradient vector.
  - None means that no gradient information is used or estimated. Not available if optimizer is in ["lm", "trf", "dogbox"].

- **bounded**
  (bool, default False) – If True, performs bounded parameter optimization if supported by optimizer.

- **update_plot**
  (bool, default False) – If True, the plot is updated during the optimization process. It slows down the optimization, but it enables visualization of the optimization progress.

- **print_info**
  (bool, default False) – If True, print information about the fitting results, which are also stored in `model.fit_output` in the form of a `scipy.optimize.OptimizeResult` object.

- **return_info**
  (bool, default True) – If True, returns the fitting results in the form of a `scipy.optimize.OptimizeResult` object.

- **fd_scheme**
  (str {"2-point", "3-point", "cs"}, default "2-point") – If grad='fd', selects the finite difference scheme to use. See `scipy.optimize.minimize()` for details. Ignored if optimizer is "lm", "trf" or "dogbox".
• **kwargs (keyword arguments) – Any extra keyword argument will be passed to the chosen optimizer. For more information, read the docstring of the optimizer of your choice in scipy.optimize.

Returns

 Return type  None

Notes

The chi-squared and reduced chi-squared statistics, and the degrees of freedom, are computed automatically when fitting, only when loss_function="ls". They are stored as signals: chisq, red_chisq and dof.

If the attribute metada.Signal.Noise_properties.variance is defined as a Signal instance with the same navigation_dimension as the signal, and loss_function is "ls" or "huber", then a weighted fit is performed, using the inverse of the noise variance as the weights.

Note that for both homoscedastic and heteroscedastic noise, if metadata.Signal.Noise_properties.variance does not contain an accurate estimation of the variance of the data, then the chi-squared and reduced chi-squared statistics will not be be computed correctly. See the Setting the noise properties in the User Guide for more details.

See also:

• multifit()
• fit()

**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.

**insert** (**kwargs)

Insert object before index.

**load_parameters_from_file** (filename)

Loads the parameters array from a binary file written with the ‘save_parameters2file’ function.

Parameters filename (str) –

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.

```python
multifit(mask=None, fetch_only_fixed=False, autosave=False, autosave_every=10, show_progressbar=None, interactive_plot=False, iterpath=None, **kwargs)
```

Fit the data to the model at all positions of the navigation dimensions.

**Parameters**

- `mask (np.ndarray, optional)` – To mask (i.e. do not fit) at certain position, pass a boolean numpy.array, where True indicates that the data will NOT be fitted at the given position.

- `fetch_only_fixed (bool, default False)` – If True, only the fixed parameters values will be updated when changing the position.

- `autosave (bool, default False)` – If True, the result of the fit will be saved automatically with a frequency defined by autosave_every.

- `autosave_every (int, default 10)` – Save the result of fitting every given number of spectra.

- `show_progressbar (None or bool)` – If True, display a progress bar. If None, the default from the preferences settings is used.

- `interactive_plot (bool, default False)` – If True, update the plot for every position as they are processed. Note that this slows down the fitting by a lot, but it allows for interactive monitoring of the fitting (if in interactive mode).

- `iterpath (None, "flyback", "serpentine"), default None)` –
  
  If “flyback”: At each new row the index begins at the first column, in accordance with the way `numpy.ndindex` generates indices.

  If “serpentine”: Iterate through the signal in a serpentine, “snake-game”-like manner instead of beginning each new row at the first index. Works for n-dimensional navigation space, not just 2D.

  If None: Currently None -> "flyback". The default argument will use the "flyback" iterpath, but shows a warning that this will change to "serpentine" in version 2.0.

- `**kwargs (keyword arguments)` – Any extra keyword argument will be passed to the fit method. See the documentation for `fit()` for a list of valid arguments.

**Returns**

**Return type** None

**See also:**

- `fit()`

```python
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.

**Parameters**

- **only_free** *(bool)* – If True, only components with free parameters will be printed. Within these, only parameters which are free will be printed.
- **only_active** *(bool)* – If True, only values of active components will be printed
- **component_list** *(None or list of 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
>>> 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)
```

**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(bounded=True)

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(bounded=True)

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
>>> v1 = hs.model.components1D.Voigt()
>>> m.append(v1)
>>> m.set_parameters_not_free()

>>> m.set_parameters_not_free(component_list=[v1],
                          parameter_name_list=['area','centre'])
```

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
>>> v1 = hs.model.components1D.Voigt()
>>> v2 = hs.model.components1D.Voigt()
>>> m.extend([v1, v2])
>>> m.set_parameters_value('area', 5)
>>> m.set_parameters_value('area', 5, component_list=[v1])
>>> m.set_parameters_value('area', 5, component_list=[v1],
                       only_current=True)
```

**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** *( )* 

Stores 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.model._check_deprecated_optimizer(optimizer)*

Can be removed in HyperSpy 2.0

**hyperspy.model.reconstruct_component(comp_dictionary, **init_args)**

**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:

_set_widget_type()  _apply_roi2widget(widget)  _set_from_widget(widget)

Sets up events.changed event, and inits HasTraits.

_apply_roi2widget (widget)
This function is responsible for applying the ROI geometry to the widget. When this function is called, the
widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

_get_widget_type (axes, signal)
Get the type of a widget that can represent the ROI on the given axes and signal.

_on_widget_change (widget)
Callback for widgets’ ‘changed’ event. Updates the internal state from the widget, and triggers events
(excluding connections to the source widget).

_remove_widget (widget)

_set_from_widget (widget)
Sets the internal representation of the ROI from the passed widget, without doing anything to events.

_update_widgets (exclude=None)
Internal function for updating the associated widgets to the geometry contained in the ROI.

Parameters

exclude (set()) – A set of widgets to exclude from the update. Useful e.g. if
a widget has triggered a change in the ROI: Then all widgets, excluding the one that was the
source for the change, should be updated.

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.

Parameters

• signal (Signal) – The signal to witch 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 ar-
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 for any widget passed with the ‘widget’ argument.

• **kwargs** – All keyword argument are passed to the widget constructor.

```python
interactive (signal, navigation_signal='same', out=None, color='green', **kwargs)
```

Creates an interactively sliced Signal (sliced by this ROI) via hyperspy.interactive.

**Parameters**

• **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 for 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.

```python
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.

```python
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.

Sets up events.changed event, and inits HasTraits.

```python
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.

```python
_get_ranges ()
```

Utility to get the value ranges that the ROI would select.

If the ROI is point base or is rectangluar in nature, these can be used to slice a signal. Extracted from _make_slices() to ease implementation in inherited ROIs.

```python
_make_slices (axes_collection, axes, ranges=None)
```

Utility function to make a slice structure that will slice all the axes in ‘axes_collection’. The axes in the axes argument will be sliced by the ROI, all other axes with ‘slice(None)’. Alternatively, if ‘ranges’ is passed, axes[i] will be sliced with ‘ranges[i]’.

```python
_ndim = 0
```
_parse_axes(axes, axes_manager)

Utility function to parse the ‘axes’ argument to a list of DataAxis.

Parameters

• axes (specification of axes to use, default is None) – The axes argument specifies which axes the ROI will be applied on. The axes 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 the signal AxesManager
  – 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.

• axes_manager (AxesManager) – The AxesManager to use for parsing axes, if axes is not already a tuple of DataAxis.

Returns

Return type DataAxis

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.

class hyperspy.roi.CircleROI(cx, cy, r, r_inner=None)
Bases: hyperspy.roi.BaseInteractiveROI

Selects a circular or annular region in a 2D space. The coordinates of the center of the circle are stored in the ‘cx’ and ‘cy’ attributes. The radious in the r attribute. If an internal radious is defined using the r_inner attribute, then an annular region is selected instead. CircleROI can be used in place of a tuple containing (cx, cy, r), (cx, cy, r, r_inner) when r_inner is not None.

Sets up events.changed event, and inits HasTraits.

_apply_roi2widget(widget)

This function is responsible for applying the ROI geometry to the widget. When this function is called, the widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

(cx_changed(old, new)

(cy_changed(old, new)

_get_widget_type(axes, signal)

Get the type of a widget that can represent the ROI on the given axes and signal.

(ndim = 2

(r_changed(old, new)

(r_inner_changed(old, new)

_set_from_widget(widget)

Sets the internal representation of the ROI from the passed widget, without doing anything to events.
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.Line2DROI(x1, y1, x2, y2, linewidth=0)
Bases: hyperspy.roi.BaseInteractiveROI

Selects a line of a given width in 2D space. The coordinates of the end points of the line are stored in the
x1, y1, x2, y2 attributes. The length is available in the length attribute and the method angle computes the angle of the
line with the axes.

Line2DROI can be used in place of a tuple containing the coordinates of the two end-points of the line and the
linewidth (x1, y1, x2, y2, linewidth).

Sets up events.changed event, and inits HasTraits.

_apply_roi2widget(widget)
This function is responsible for applying the ROI geometry to the widget. When this function is called, the
widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

_get_widget_type(axes, signal)
Get the type of a widget that can represent the ROI on the given axes and signal.

static _line_profile_coordinates(src, dst, linewidth=1)
Return the coordinates of the profile of an image along a scan line.

Parameters

- **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

Returns coords – The coordinates of the profile along the scan line. The length of the profile
is the ceil of the computed length of the scan line.

Return type array, shape (2, N, C), float
Notes

This is a utility method meant to be used internally by skimage functions. The destination point is included in the profile, in contrast to standard numpy indexing.

_def_width_changed(old, new)

_set_from_widget(widget)

Sets the internal representation of the ROI from the passed widget, without doing anything to events.

_x1_changed(old, new)

_x2_changed(old, new)

_y1_changed(old, new)

_y2_changed(old, new)

_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., 1)
>>> r.angle()
63.43494882292201
```

gui(display=True, toolkit=None, **kwargs)

Display or return interactive GUI element if available.

Parameters

- display (bool) – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.
- toolkit (str, iterable of 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 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.

_Point1DROI_ can be used in place of a tuple containing the value of _value_.

604 Chapter 3. hyperspy
Example

```python
>>> roi = hs.roi.Point1DROI(0.5)
>>> value, = roi
>>> print(value)
0.5
```

Sets up events.changed event, and inits HasTraits.

```python
_apply_roi2widget(widget)
```

This function is responsible for applying the ROI geometry to the widget. When this function is called, the widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

```python
_get_ranges()
```

Utility to get the value ranges that the ROI would select.

If the ROI is point base or is rectangular in nature, these can be used to slice a signal. Extracted from _make_slices() to ease implementation in inherited ROIs.

```python
_get_widget_type(axes, signal)
```

Get the type of a widget that can represent the ROI on the given axes and signal.

```python
_ndim = 1
```

```python
_set_from_widget(widget)
```

Sets the internal representation of the ROI from the passed widget, without doing anything to events.

```python
_value_changed(old, new)
```

`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
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.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’.

`Point2DROI` can be used in place of a tuple containing the coordinates of the point `(x, y)`.
Example

```python
>>> roi = hs.roi.Point2DROI(3, 5)
>>> x, y = roi
>>> print(x, y)
3 5
```

Sets up events.changed event, and inits HasTraits.

```python
_apply_roi2widget(widget)
```

This function is responsible for applying the ROI geometry to the widget. When this function is called, the widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

```python
_get_ranges()
```

Utility to get the value ranges that the ROI would select.

If the ROI is point base or is rectangular in nature, these can be used to slice a signal. Extracted from _make_slices() to ease implementation in inherited ROIs.

```python
_get_widget_type(axes, signal)
```

Get the type of a widget that can represent the ROI on the given axes and signal.

```python
_ndim = 2
```

```python
_set_from_widget(widget)
```

Sets the internal representation of the ROI from the passed widget, without doing anything to events.

```python
_x_changed(old, new)
```

```python
_y_changed(old, new)
```

```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
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.RectangularROI(left, top, right, bottom)
```

Selects a range in a 2D space. The coordinates of the range in the 2D space are stored in the traits ‘left’, ‘right’, ‘top’ and ‘bottom’. Convenience properties ‘x’, ‘y’, ‘width’ and ‘height’ are also available, but cannot be used for initialization.

RectangularROI can be used in place of a tuple containing (left, right, top, bottom).
Example

```python
>>> roi = hs.roi.RectangularROI(left=0, right=10, top=20, bottom=20.5)
>>> left, right, top, bottom = roi
>>> print(left, right, top, bottom)
0 10 20 20.5
```

Sets up events.changed event, and inits HasTraits.

```python
_apply_roi2widget(widget)
```
This function is responsible for applying the ROI geometry to the widget. When this function is called, the widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

```python
_bottom_changed(old, new)
```

```python
_get_ranges()
```
Utility to get the value ranges that the ROI would select.

If the ROI is point base or is rectangular in nature, these can be used to slice a signal. Extracted from _make_slices() to ease implementation in inherited ROIs.

```python
_get.widget_type(axes, signal)
```
Get the type of a widget that can represent the ROI on the given axes and signal.

```python
_left_changed(old, new)
```

```python
_ndim = 2
```

```python
_right_changed(old, new)
```

```python
_set_from.widget(widget)
```
Sets the internal representation of the ROI from the passed widget, without doing anything to events.

```python
_top_changed(old, new)
```

```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.

**property height**

Returns / sets the height of the ROI

```python
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’.

*SpanROI* can be used in place of a tuple containing the left and right values.

**Example**

```python
>>> roi = hs.roi.SpanROI(-3, 5)
>>> left, right = roi
>>> print(left, right)
3 5
```

Sets up the `events.changed` event, and initiates HasTraits.

**_apply_roi2widget (widget)**

This function is responsible for applying the ROI geometry to the widget. When this function is called, the widget’s events are already suppressed, so this should not be necessary for _apply_roi2widget to handle.

**_get_ranges ()**

Utility to get the value ranges that the ROI would select.

If the ROI is point base or is rectangular in nature, these can be used to slice a signal. Extracted from _make_slices() to ease implementation in inherited ROIs.

**_get_widget_type (axes, signal)**

Get the type of a widget that can represent the ROI on the given axes and signal.

**_left_changed (old, new)**

**_ndim = 1**

**_right_changed (old, new)**

**_set_from_widget (widget)**

Sets the internal representation of the ROI from the passed widget, without doing anything to events.

**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._get_mpl_ax(plot, axes)

Returns MPL Axes that contains the axes.

The space of the first DataAxis in axes will be used to determine which plot’s matplotlib Axes to return.

Parameters

- **plot** (*MPL_HyperExplorer*) – The explorer that contains the navigation and signal plots
- **axes** (*collection of DataAxis*) – The axes to infer from.

hyperspy.roi.guess_vertical_or_horizontal(axes, signal)

**hyperspy.samfire module**

**class hyperspy.samfire.Samfire**(model, workers=None, setup=True, random_state=None, **kwargs)

Bases: object

Smart Adaptive Multidimensional Fitting (SAMFire) object

SAMFire is a more robust way of fitting multidimensional datasets. By extracting starting values for each pixel from already fitted pixels, SAMFire stops the fitting algorithm from getting lost in the parameter space by always starting close to the optimal solution.

SAMFire only picks starting parameters and the order the pixels (in the navigation space) are fitted, and does not provide any new minimisation algorithms.

**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 histograms 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

random_state
Random seed used to select the next pixels.
Type None or int or RandomState instance, default None

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

_Samfire__active_strategy_ind = 0

property _active_strategy_ind
_args = None
HyperSpy Documentation, Release 1.6.0

```python

__enable_optional_components__()
__figure__ = None
__next_pixels__(number)
__progressbar__ = None
__request_user_input__()
__run_active_strategy__()
__run_active_strategy_one__()
__setup__(**kwargs)
    Set up SAMFire - configure models, set up pool if necessary
__swap_dict_and_model__(m_ind, dict_, d_ind=None)
__workers__ = None

@property
def 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 diffu-
    sion 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
def pixels_done
    Returns the number of pixels that have been solved
```

3.1. hyperspy package 611
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
stop ()
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 (samf)
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(signal)
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.

_additional_slicing_targets = ['metadata.Signal.Noise_properties.variance']
_alias_signal_types = []
_apply_function_on_data_and_remove_axis(function, axes, out=None, **kwargs)
_assign_subclass()
_binary_operator_ruler(other, op_name)
_calculate_summary_statistics(**kwargs)
_check_signal_dimension_equals_one()
_check_signal_dimension_equals_two()
_create_metadata()
_cycle_signal()
Cycles over the signal data.

It is faster than using the signal iterator.

Warning! could produce a infinite loop.

property _data_aligned_with_axes
Returns a view of `data` with is axes aligned with the Signal axes.

_deepcopy_with_new_data(data=None, copy_variance=False)
Returns a deepcopy of itself replacing the data.
This method has an advantage over the default `copy.deepcopy()` in that it does not copy the data, which can save memory.

**Parameters**

- **data** (None or `numpy.ndarray`)
- **copy_variance** (`bool`) – Whether to copy the variance of the signal to the new copy

**Returns** `ns` – The newly copied signal

**Return type** `BaseSignal` (or subclass)

```python
_dtype = 'real'
```

```python
static _estimate_poissonian_noise_variance(dc, gain_factor, gain_offset, correlation_factor)
```

```python
_get_navigation_signal(data=None, dtype=None)
```

Return a signal with the same axes as the navigation space.

**Parameters**

- **data** (None or `numpy.ndarray`, optional) – If None, the resulting Signal data is an array of the same `dtype` as the current one filled with zeros. If a numpy array, the array must have the correct dimensions.
- **dtype** (`numpy.dtype`, optional) – The desired data-type for the data array when `data` is None, e.g., `numpy.int8`. The default is the data type of the current signal data.

```python
_get_signal_signal(data=None, dtype=None)
```

Return a signal with the same axes as the signal space.

**Parameters**

- **data** (None or `numpy.ndarray`, optional) – If None, the resulting Signal data is an array of the same `dtype` as the current one filled with zeros. If a numpy array, the array must have the correct dimensions.
- **dtype** (`numpy.dtype`, optional) – The desired data-type for the data array when `data` is None, e.g., `numpy.int8`. The default is the data type of the current signal data.

```python
_get_undefined_axes_list()
```

```python
_iterate_signal()
```

It is faster than using the signal iterator.

```python
_lazy = False
```

```python
_load_dictionary(file_data_dict)
```

Load data from dictionary.

**Parameters** `file_data_dict` (`dict`) – A dictionary containing at least a ‘data’ keyword with an array of arbitrary dimensions. Additionally the dictionary can contain the following items:

- **data**: the signal data. It can be an array of any dimensions.
- **axes**: a dictionary to define the axes (see the documentation of the `AxesManager` class for more details).
• attributes: a dictionary whose items are stored as attributes.
• metadata: 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: 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.

```python
_ma_workaround(s, function, axes, ar_axes, out)
_make_sure_data_is_contiguous()
_map_all (function, inplace=True, **kwargs)
    The function has to have either 'axis' or 'axes' keyword argument, and hence support operating on the full dataset efficiently.
    Replaced for lazy signals
_map_iterate (function, iterating_kwargs=(), show_progressbar=None, parallel=None, max_workers=None, ragged=None, inplace=True, **kwargs)
    Iterates the signal navigation space applying the function.
```

**Parameters**

- **function** (function) – the function to apply
- **iterating_kwargs** (tuple (of tuples)) – A tuple with structure (‘key1’, value1), (‘key2’, value2), ..) where the key-value pairs will be passed as kwargs for the function to be mapped, and the values will be iterated together with the signal navigation.
- **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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the max_workers argument.
- **max_workers** (None or int) – Maximum number of threads used when parallel=True. If None, defaults to \(\min(32, \text{os.cpu_count}())\).
- **inplace** (bool, default True) – If True, the data is replaced by the result. Otherwise a new signal with the results is returned.
- **ragged** (None or bool, default None) – Indicates if results for each navigation pixel are of identical shape (and/or numpy arrays to begin with). If None, an appropriate choice is made while processing. Note: None is not allowed for Lazy signals!
- ****kwargs** (dict) – Additional keyword arguments passed to function
Notes

This method is replaced for lazy signals.

Examples

Pass a larger array of different shape

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((20,1)))
>>> def func(data, value=0):
...     return data + value
>>> # pay attention that it's a tuple of tuples - need commas
>>> s._map_iterate(func,
...     iterating_kwargs=(('value',
...         np.random.rand(5,400).flat),))
```

storing function result to other signal (e.g. calculated shifts)

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((5,4)))
>>> def func(data):
...     # the original function
...     return data.sum()  
>>> result = s._get_navigation_signal().T
>>> def wrapped(*args, data=None):
...     return func(data)
>>> result._map_iterate(wrapped,
...     iterating_kwargs=(('data', s),))
```

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((20,1)))
>>> def func(data, value=0):
...     return data + value
>>> # pay attention that it's a tuple of tuples - need commas
>>> s._map_iterate(func,
...     iterating_kwargs=(('value',
...         np.random.rand(5,400).flat),))
```

Storing function result to other signal (e.g. calculated shifts)

```python
>>> s = hs.signals.Signal1D(np.arange(20.).reshape((5,4)))
>>> def func(data):
...     # the original function
...     return data.sum()  
>>> result = s._get_navigation_signal().T
>>> def wrapped(*args, data=None):
...     return func(data)
>>> result._map_iterate(wrapped,
...     iterating_kwargs=(('data', s),))
```
unary_operator_ruler(op_name)

unfold(steady_axes, unfolded_axis)

Modify the shape of the data by specifying the axes whose dimension do not change and the axis over which the remaining axes will be unfolded.

Parameters

• steady_axes(list) – The indices of the axes which dimensions do not change.

• unfolded_axis(int) – The index of the axis over which all the rest of the axes (except the steady axes) will be unfolded.

See also:

fold()

Notes

WARNING: this private function does not modify the signal subclass and it is intended for internal use only. To unfold use the public unfold(), unfold_navigation_space(), unfold_signal_space() instead. It doesn’t make sense to perform an unfolding when \( \text{dim} < 2 \)

validate_rebin_args_and_get_factors(new_shape=None, scale=None)

add_gaussian_noise(std, random_state=None)

Add Gaussian noise to the data.

The operation is performed in-place (i.e. the data of the signal is modified). This method requires the signal to have a float data type, otherwise it will raise a TypeError.

Parameters

• std(float) – The standard deviation of the Gaussian noise.

• random_state (None or int or RandomState instance, default None) – Seed for the random generator.

Note: This method uses numpy.random.normal() (or dask.array.random.normal() for lazy signals) to generate the noise.

add_marker(marker, plot_on_signal=True, plot_marker=True, permanent=False, plot_signal=True, render_figure=True)

Add one or several markers to the signal or navigator plot and plot the signal, if not yet plotted (by default).

Parameters

• marker(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.random((2, 3, 10)))
>>> marker = hs.markers.vertical_line([[1, 3, 5], [2, 4, 6]])
>>> s.add_marker(marker, permanent=True)
```

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, 100)))
>>> marker = hs.markers.point((10, 30, 50), (30, 50, 60), color='red')
>>> s.add_marker(marker, permanent=True, plot_marker=False)
```

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, random_state=None)*

Add Poissonian noise to the data.

This method works in-place. The resulting data type is **int64**. If this is different from the original data type then a warning is added to the log.

**Parameters**

- **keep_dtype** *(bool, default True)* – If True, keep the original data type of the signal data. For example, if the data type was initially 'float64', the result of the operation (usually 'int64') will be converted to 'float64'.
- **random_state** *(None or int or RandomState instance, default None)* – Seed for the random generator.

**Note:** This method uses `numpy.random.poisson()` (or `dask.array.random.poisson()` for lazy signals) to generate the Poissonian noise.

**apply_apodization** *(window='hann', hann_order=None, tukey_alpha=0.5, inplace=False)*

Apply an apodization window to a Signal.

**Parameters**

- **window** *(str, optional)* – Select between {hann (default), 'hamming', or 'tukey'}
- **hann_order** *(None or int, optional)* – Only used if window='hann'
  If integer \(n\) is provided, a Hann window of \(n\)-th order will be used. If None, a first order Hann window is used. Higher orders result in more homogeneous intensity distribution.
- **tukey_alpha** *(float, optional)* – Only used if window='tukey' (default is 0.5). From the documentation of `scipy.signal.windows.tukey()`:
  - Shape parameter of the Tukey window, representing the fraction of the window inside the cosine tapered region. If zero, the Tukey window is equivalent to a rectangular window. If one, the Tukey window is equivalent to a Hann window.
- **inplace** *(bool, optional)* – If True, the apodization is applied in place, i.e. the signal data will be substituted by the apodized one (default is False).

**Returns**

- **out** – 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  \texttt{LazySignal}

\texttt{as\_signal1D}(\texttt{spectral\_axis, out=\texttt{None}}, \texttt{optimize=\texttt{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 \texttt{transpose()} for a more general method with more options.

\textbf{Parameters}

- \texttt{spectral\_axis} (\texttt{int}, \texttt{str}, or \texttt{DataAxis}) – The axis can be passed directly, or specified using the index of the axis in the Signal’s \texttt{axes\_manager} or the axis name.
- \texttt{out} (\texttt{BaseSignal} (or subclasses) or \texttt{None}) – If \texttt{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.
- \texttt{optimize} (\texttt{bool}) – If \texttt{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 \textit{Transposing (changing signal spaces)} section of the HyperSpy user guide for more information. When operating on lazy signals, if \texttt{True}, the chunks are optimised for the new axes configuration.

\textbf{See also:}
\texttt{as\_signal2D()}, \texttt{transpose()}, \texttt{hyperspy.misc.utils.transpose()}

\textbf{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)>
```

\texttt{as\_signal2D}(\texttt{image\_axes, out=\texttt{None}}, \texttt{optimize=\texttt{True}})

Convert a signal to image (\texttt{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.

\textbf{Parameters}

- \texttt{image\_axes} (tuple (of \texttt{int}, \texttt{str} or \texttt{DataAxis})) – Select the image axes. Note that the order of the axes matters and it is given in the “natural” i.e. \texttt{X, Y, Z...} order.
- \texttt{out} (\texttt{BaseSignal} (or subclasses) or \texttt{None}) – If \texttt{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.
- \texttt{optimize} (\texttt{bool}) – If \texttt{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 \textit{Transposing (changing signal spaces)} section of the HyperSpy user guide for more information. When operating on lazy signals, if \texttt{True}, the chunks are optimised for the new axes configuration.
Raisers **DataDimensionError** – When \(data.ndim < 2\)

See also:

`as_signal1D()`, `transpose()`, `hyperspy.misc.utils.transpose()`

**Examples**

```python
>>> s = hs.signals.Signal1D(np.ones((2,3,4,5)))
>>> s
<Signal1D, title: , dimensions: (4, 3, 2, 5)>
>>> s.as_signal2D((0,1))
<Signal2D, title: , dimensions: (5, 2, 4, 3)>
```

```python
>>> s.to_signal2D((1,2))
<Signal2D, title: , dimensions: (4, 5, 3, 2)>
```

**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)8 or rgb(a)16, and the navigation_dimension should be at least 2. After conversion, the signal_dimension becomes 2. The dtype of images with original dtype rgb(a)8 or rgb(a)16 can only be changed to `uint8` or `uint16`, and the signal_dimension becomes 1.

- **rechunk** (bool) – Only has effect when operating on lazy signal. 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.])
```

**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 gain_factor, \(b\) is the gain_offset (the Gaussian noise variance) and \(c\) the correlation_factor. The correlation factor accounts for correlation of adjacent signal elements that can be modeled as a convolution with a Gaussian point spread function.

Parameters

• expected_value (None or BaseSignal (or subclasses)) – If None, the signal data is taken as the expected value. Note that this may be inaccurate where the value of data is small.

• gain_factor (None or float) – \(a\) in the above equation. Must be positive. If None, take the value from metadata.Signal.Noise_properties.Variance_linear_model if defined. Otherwise, suppose
pure Poissonian noise (i.e. gain_factor=1). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

- **gain_offset** *(None or float)* – $b$ in the above equation. Must be positive. If None, take the value from `metadata.Signal.Noise_properties.Variance_linear_model` if defined. Otherwise, suppose pure Poissonian noise (i.e. gain_offset=0). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

- **correlation_factor** *(None or float)* – $c$ in the above equation. Must be positive. If None, take the value from `metadata.Signal.Noise_properties.Variance_linear_model` if defined. Otherwise, suppose pure Poissonian noise (i.e. correlation_factor=1). If not None, the value is stored in `metadata.Signal.Noise_properties.Variance_linear_model`.

### fft *(shift=False, apodization=False, real_fft_only=False, **kwargs)*

Compute the discrete Fourier Transform.

This function computes the discrete Fourier Transform over the signal axes by means of the Fast Fourier Transform (FFT) as implemented in numpy.

**Parameters**

- **shift** *(bool, optional)* – If True, the origin of FFT will be shifted to the centre (default is False).

- **apodization** *(bool or str)* – Apply an apodization window before calculating the FFT in order to suppress streaks. Valid string values are {'hann' or 'hamming' or 'tukey'} If True or 'hann', applies a Hann window. If 'hamming' or 'tukey', applies Hamming or Tukey windows, respectively (default is False).

- **real_fft_only** *(bool, default False)* – If True and data is real-valued, uses `numpy.fft.rfftn()` instead of `numpy.fft.fftn()`

- ****kwargs** *(dict)* – other keyword arguments are described in `numpy.fft.fftn()`

**Returns** *s* – A Signal containing the result of the FFT algorithm

**Return type** *ComplexSignal*

**Examples**

```python
>>> im = hs.signals.Signal2D(scipy.misc.ascent())
>>> im.fft()
<ComplexSignal2D, title: FFT of , dimensions: (512, 512)>
```

```python
>>> # 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 tmp_parameters.filename is defined (which is always the case when the Signal has been read from a file), the filename stored in the metadata is modified by appending an underscore and the current indices in parentheses.

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 = hs.signals.Signal2D(np.zeros((2, 3, 32, 32)))
>>> im
<Signal2D, title: , dimensions: (3, 2, 32, 32)>
>>> im.axes_manager.indices = 2, 1
>>> im.get_current_signal()
<Signal2D, title: (2, 1), dimensions: (32, 32)>
```

get_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='fd', range_bins=None, max_num_bins=250, out=None, **kwargs)
Return a histogram of the signal data.

More sophisticated algorithms for determining the bins can be used by passing a string as the bins argument. Other than the 'blocks' and 'knuth' methods, the available algorithms are the same as numpy.histogram().

Note: The lazy version of the algorithm only supports "scott" and "fd" as a string argument for bins.

Parameters

• bins (int or sequence of scalars or str, default "fd") – If bins is an int, it defines the number of equal-width bins in the given range. If bins is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If bins is a string from the list below, will use the method chosen to calculate the optimal bin width and consequently the number of bins (see Notes for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the ‘auto’ option is suggested. Weighted data is not supported for automated bin size selection.

'auto' Maximum of the ‘sturges’ and ‘fd’ estimators. Provides good all around performance.
'fd' (Freedman Diaconis Estimator) Robust (resilient to outliers) estimator that takes into account data variability and data size.

'doane' An improved version of Sturges’ estimator that works better with non-normal datasets.

'scott' Less robust estimator that that takes into account data variability and data size.

'stone' Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott’s rule.

'rice' Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

'sturges’ R’s default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.

'sqrt’ Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.

'knuth’ Knuth’s rule is a fixed-width, Bayesian approach to determining the optimal bin width of a histogram.

'blocks’ Determination of optimal adaptive-width histogram bins using the Bayesian Blocks algorithm.

• range_bins (tuple or None, optional) – the minimum and maximum range for the histogram. If range_bins is None, (x.min(), x.max()) will be used.

• max_num_bins (int, default 250) – When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a MemoryError being raised by numpy.histogram().

• out (BaseSignal (or 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
• numpy.histogram()
• dask.histogram()
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()
```

going_noise_variance()

Get the noise variance of the signal, if set.

Equivalent to `s.metadata.Signal.Noise_properties.variance`.

**Parameters** None

**Returns** variance – Noise variance of the signal, if set. Otherwise returns None.

**Return type** None or float or `BaseSignal` (or subclasses)

ifft(shift=None, return_real=True, **kwargs)

Compute the inverse discrete Fourier Transform.

This function computes the real part of the inverse of the discrete Fourier Transform over the signal axes by means of the Fast Fourier Transform (FFT) as implemented in numpy.

**Parameters**

- **shift** (bool or None, optional) – If None, the shift option will be set to the original status of the FFT using the value in metadata. If no FFT entry is present in metadata, the parameter will be set to False. If True, the origin of the FFT will be shifted to the centre. If False, the origin will be kept at (0, 0) (default is None).

- **return_real** (bool, default True) – If True, returns only the real part of the inverse FFT. If False, returns all parts.

- ****kwargs (dict) – other keyword arguments are described in `numpy.fft.ifftn()`

**Returns** s – 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()`

indexmax(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**

```python
map(function, show_progressbar=None, parallel=None, max_workers=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-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 multithreading. If None, the default from the preferences settings is used. The number of threads is controlled by the `max_workers` argument.
- **max_workers** *(None or int)* – Maximum number of threads used when `parallel=True`. If None, defaults to `min(32, os.cpu_count())`.
- **inplace** *(bool, default True)* – if True, the data is replaced by the result. Otherwise a new Signal with the results is returned.
- **ragged** *(None or bool, default None)* – Indicates if the results for each navigation pixel are of identical shape (and/or numpy arrays to begin with). If None, the appropriate choice is made while processing. If True in case of lazy signal, the signal will be compute at the end of the mapping. 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{\texttt{None}}, \text{\texttt{out}}=\text{\texttt{None}}, \text{\texttt{rechunk}}=\text{\texttt{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).

- **\text{\texttt{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.

- **\text{\texttt{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{\texttt{None}}, \text{\texttt{out}}=\text{\texttt{None}}, \text{\texttt{rechunk}}=\text{\texttt{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).

- **\text{\texttt{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.

- **\text{\texttt{rechunk}}** *(bool)* – Only has effect when operating on lazy signal. If True (default), the data may be automatically rechunked before performing this operation.
Returns

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)) –
  - 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 \( \text{navigation\_dimension} > 1 \), 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 inte-
grating 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
\( \text{s.add_marker(marker, permanent=True)} \). Note, a large number of markers might lead
to very slow plotting.

• navigator_kwds (dict) – Only for image navigator, additional keyword argu-
ments for matplotlib.pyplot.imshow().

• 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).

• autoscale (str) – The string must contain any combination of the ‘x’ and ‘v’
characters. If ‘x’ or ‘v’ (for values) are in the string, the corresponding horizontal or
vertical axis limits are set to their maxima and the axis limits will reset when the data
or the navigation indices are changed. Default is ‘v’.

• **kwargs – Only when plotting an image: 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
```
**rollaxis** *(axis, to_axis, optimize=False)*

Roll the specified axis backwards, until it lies in a given position.

**Parameters**

- **axis** *(int, str, or DataAxis)* – The axis can be passed directly, or specified using the index of the axis in the Signal’s `axes_manager` or the axis name. The axis to roll backwards. The positions of the other axes do not change relative to one another.

- **to_axis** *(int, str, or DataAxis)* – The axis can be passed directly, or specified using the index of the axis in the Signal’s `axes_manager` or the axis name. The axis is rolled until it lies before this other axis.

- **optimize** *(bool)* – If True, the location of the data in memory is optimised for the fastest iteration over the navigation axes. This operation can cause a peak of memory usage and requires considerable processing times for large datasets and/or low specification hardware. See the Transposing (changing signal spaces) section of the HyperSpy user guide for more information. When operating on lazy signals, if True, the chunks are optimised for the new axes configuration.

**Returns** s – 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_noise_variance**(variance)

Set the noise variance of the signal.

Equivalent to `s.metadata.set_item("Signal.Noise_properties.variance", variance)`.

**Parameters** variance *(None or float or BaseSignal (or subclasses))* – Value or values of the noise variance. A value of None is equivalent to clearing the variance.

**Returns**

**Return type** None

**set_signal_origin**(origin)

Set the `signal_origin` metadata value.

The `signal_origin` attribute specifies if the data was obtained through experiment or simulation.

**Parameters** origin *(str)* – Typically 'experiment' or 'simulation'

**set_signal_type**(signal_type='')

Set the signal type and convert the current signal accordingly.

The `signal_type` attribute specifies the type of data that the signal contains e.g. electron energy-loss spectroscopy data, photoemission spectroscopy data, etc.

When setting `signal_type` to a “known” type, HyperSpy converts the current signal to the most appropriate `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, call `hyperspy.utils.print_known_signal_types()`, and see the developer guide for details on how to add new signal_types. A non-exhaustive list of HyperSpy extensions is also maintained here: https://github.com/hyperspy/hyperspy-extensions-list.

**Parameters** signal_type *(str, optional)* – If no arguments are passed, the `signal_type` is set to undefined and the current signal converted to a generic signal subclass. Otherwise, set the `signal_type` to the given signal type or to the signal type corresponding to the given signal type alias. Setting the `signal_type` to a known signal type...
(if exists) is highly advisable. If none exists, it is good practice to set signal_type to a
value that best describes the data signal type.

See also:

- \texttt{hyperspy.utils.print_known_signal_types()}

Examples

Let’s first print all known signal types:

```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 \texttt{signal_type} using the \texttt{signal_type}:

```python
>>> s.set_signal_type("EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
>>> s.set_signal_type("EDS_SEM")
>>> s
<EDSSEMSpectrum, title: , dimensions: (|4)>
```

or any of its aliases:

```python
>>> s.set_signal_type("TEM EELS")
>>> s
<EELSSpectrum, title: , dimensions: (|4)>
```

To set the \texttt{signal_type} to \textit{undefined}, simply call the method without arguments:

```python
>>> s.set_signal_type()
>>> s
<Signal1D, title: , dimensions: (|4)>
```

\texttt{split} (\texttt{axis='auto', number_of_parts='auto', step_sizes='auto'})

Splits the data into several signals.

The split can be defined by giving the \texttt{number_of_parts}, a homogeneous step size, or a list of customized step sizes. By default ('auto'), the function is the reverse of \texttt{stack()}.

Parameters

- \texttt{axis (int, str, or DataAxis)} – The axis can be passed directly, or specified using the index of the axis in the Signal’s \texttt{axes_manager} or the axis name. If 'auto' and if the object has been created with \texttt{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])
[<Signal1D, title: , dimensions: (3, 1|2)>,
 <Signal1D, title: , dimensions: (3, 2|2)>]
```

**Returns** splitted – 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:

\texttt{max()}, \texttt{min()}, \texttt{sum()}, \texttt{mean()}, \texttt{var()}, \texttt{indexmax()}, \texttt{indexmin()}, \texttt{valuemax()}, \texttt{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)
```

\texttt{sum}(axis=\texttt{None}, \texttt{out}=\texttt{None}, \texttt{rechunk}=\texttt{True})

Sum the data over the given axes.

**Parameters**

- \texttt{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 \texttt{axes_manager} or the name of the axis. Any duplicates are removed. If \texttt{None}, the operation is performed over all navigation axes (default).

- \texttt{out (BaseSignal (or subclasses) or None)} – If \texttt{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.

- \texttt{rechunk (bool)} – Only has effect when operating on lazy signal. If \texttt{True} (default), the data may be automatically rechunked before performing this operation.

**Returns** \texttt{s} – A new Signal containing the sum of the provided Signal along the specified axes.

**Return type** \texttt{BaseSignal (or subclasses)}

See also:

\texttt{max()}, \texttt{min()}, \texttt{mean()}, \texttt{std()}, \texttt{var()}, \texttt{indexmax()}, \texttt{indexmin()}, \texttt{valuemax()}, \texttt{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)
```

\texttt{swap_axes} \texttt{(axis1, axis2, optimize=False)}

Swap two axes in the signal.

**Parameters**

- \texttt{axis1 (int, str, or DataAxis)} – The axis can be passed directly, or specified using the index of the axis in the Signal’s \texttt{axes_manager} or the axis name.

- \texttt{axis2 (int, str, or DataAxis)} – The axis can be passed directly, or specified using the index of the axis in the Signal’s \texttt{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**  
- 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`, 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()`

```python
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_factors** *(factors, folder=None, comp_ids=None, multiple_files=True, save_figures=False, save_figures_format='png', factor_prefix=None, factor_format=None, comp_label=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, plot_shifts=True, plot_char=4, img_data=None, same_window=False, calibrate=True, quiver_color='white', vector_scale=1, no_nans=True, per_row=3)*

- **_export_loadings** *(loadings, folder=None, comp_ids=None, multiple_files=True, loading_prefix=None, loading_format='hspy', save_figures_format='png', comp_label=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, save_figures=False, same_window=False, calibrate=True, no_nans=True, per_row=3)*

- **_get_cluster_signals_factors** *(signal)*

- **_get_factors** *(factors)*
Plot components from PCA or ICA, or peak characteristics.

**Parameters**

- **comp_ids** *(None, int, or list of ints)*: If None, returns maps of all components. If int, returns maps of components with ids from 0 to given int. If list of ints, returns maps of components with ids in given list.

- **calibrate** *(bool)*: If True, plots are calibrated according to the data in the axes manager.

- **same_window** *(bool)*: If True, plots each factor to the same window. They are not scaled. Default True.

- **comp_label** *(str)*: Title of the plot

- **cmap** *(a matplotlib colormap)*: The colormap used for factor images or any peak characteristic scatter map overlay. Default is the matplotlib gray colormap (plt.cm.gray).

**Other Parameters**

- **img_data** *(2D numpy array)*: The array to overlay peak characteristics onto. If None, defaults to the average image of your stack.

- **plot_shifts** *(bool, default is True)*: If true, plots a quiver (arrow) plot showing the shifts for each peak present in the component being plotted.

- **plot_char** *(None or int)*: If int, the id of the characteristic to plot as the colored scatter plot. Possible components are:
  - 4: peak height
  - 5: peak orientation
  - 6: peak eccentricity

- **quiver_color** *(any color recognized by matplotlib)*: Determines the color of vectors drawn for plotting peak shifts.

- **vector_scale** *(integer or None)*: Scales the quiver plot arrows. The vector is defined as one data unit along the X axis. If shifts are small, set vector_scale so that when they are multiplied by vector_scale, they are on the scale of the image plot. If None, uses matplotlib’s autoscaling.

**Returns**

**Return type** matplotlib figure or list of figure if same_window=False

- **plot_loadings** *(loadings, comp_ids, calibrate=True, same_window=True, comp_label=None, with_factors=False, factors=None, cmap=matplotlib.colors.LinearSegmentedColormap object, no_nans=False, per_row=3, axes_decor='all')*
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 images, such as factors, loadings, or for peak characteristics. Default is the matplotlib gray colormap (plt.cm.gray).

- **per_row (int)** – The number of plots in each row, when the same_window parameter is True.
• **save_figures_format (str)** – The image format extension.

See also:

`get_bss_factors()`, `get_bss_loadings()`

**export_cluster_results**

```
export_cluster_results(cluster_ids=None, folder=None, calibrate=True, center_prefix='cluster_center',
                       center_format='hspy', membership_prefix='cluster_label', membership_format='hspy',
                       comp_label=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>,
                       same_window=False, multiple_files=True, no_nans=True, per_row=3, save_figures=False, save_figures_format='png')
```

Export results from a cluster analysis to any of the supported formats.

**Parameters**

- **cluster_ids (None, int, or list of ints)** – If None, returns all clusters/centers. If int, returns clusters/centers with ids from 0 to given int. If list of ints, returns clusters/centers with ids in given list.

- **folder (str or None)** – The path to the folder where the file will be saved. If None the current folder is used by default.

- **center_prefix (string)** – The prefix that any exported filenames for cluster centers begin with

- **center_format (string)** – The extension of the format that you wish to save to. Default is “hspy”. See loading format for more details.

- **label_prefix (string)** – The prefix that any exported filenames for cluster labels begin with

- **label_format (string)** – 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, on exporting a file per center will be created. Otherwise only two files will be created, one for the centers and another for the membership. The default value can be chosen in the preferences.

- **save_figures (bool)** – If True the same figures that are obtained when using the plot methods will be saved with 600 dpi resolution

- **options (for save_figures = True ONLY) (Plotting)** –

- **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 (string, 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** *(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_cluster_signals()`, `get_cluster_labels()`

```python
def export_decomposition_results(comp_ids=None, folder=None, calibrate=True, factor_prefix='factor', factor_format='hspy', loading_prefix='loading', loading_format='hspy', cmap=None, 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 images, such as factors, loadings, or for peak characteristics. Default is the matplotlib gray colormap (plt.cm.gray).

• **per_row**(int) – The number of plots in each row, when the same_window parameter is True.

• **save_figures_format**(str) – The image format extension.

See also:

get_decomposition_factors(), get_decomposition_loadings()

get_bss_factors()

Return the blind source separation factors.

Returns signal

Return type BaseSignal (or subclass)

See also:

get_bss_loadings(), export_bss_results()

get_bss_loadings()

Return the blind source separation loadings.

Returns signal

Return type BaseSignal (or subclass)

See also:

get_bss_factors(), export_bss_results()

get_cluster_distances()

Euclidian distances to the centroid of each cluster

See also:

get_cluster_signals()

Returns Hyperspy signal of cluster distances

Return type signal

get_cluster_labels(merged=False)

Return cluster labels as a Signal.

merged [bool] If False the cluster label signal has a navigation axes of length number_of_clusters and the signal along the the navigation direction is binary - 0 the point is not in the cluster, 1 it is included. If True, the cluster labels are merged (no navigation axes). The value of the signal at any point will be between -1 and the number of clusters. -1 represents the points that were masked for cluster analysis if any.

See also:

get_cluster_signals()
Returns

Return type  signal Hyperspy signal of cluster labels

**get_cluster_signals** *(signal='mean')*

Return the cluster centers as a Signal.

**signal**  [(“mean”, “sum”, “centroid”), optional] If “mean” or “sum” return the mean signal or sum respectively over each cluster. If “centroid”, returns the signals closest to the centroid.

See also:

**get_cluster_labels()**

**get_decomposition_factors()**

Return the decomposition factors.

Returns  signal

Return type  BaseSignal (or subclass)

See also:

**get_decomposition_loadings()**, **export_decomposition_results()**

**get_decomposition_loadings()**

Return the decomposition loadings.

Returns  signal

Return type  BaseSignal (or subclass)

See also:

**get_decomposition_factors()**, **export_decomposition_results()**

**plot_bss_factors** *(comp_ids=None, calibrate=True, same_window=True, title=None, cmap=<matplotlib.colors.LinearSegmentedColormap object>, per_row=3, **kwargs)*

Plot factors from blind source separation results. In case of 1D signal axis, each factors line can be toggled on and off by clicking on their corresponding line in the legend.

**Parameters**

- **comp_ids** *(None, int, or list of 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.

- **title** *(str)* – Title of the plot.

- **cmap** *(Colormap)* – The colormap used for the factor images, or for peak characteristics. Default is the matplotlib gray colormap (plt.cm.gray).

- **per_row** *(int)* – The number of plots in each row, when the same_window parameter is True.

See also:

**plot_bss_loadings()**, **plot_bss_results()**
**plot_bss_loadings** *(comp_ids=None, calibrate=True, same_window=True, title=None, with_factors=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', **kwargs)*

Plot loadings from blind source separation results. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on their corresponding line in the legend.

**Parameters**

- **comp_ids** *(None, int, or list (of 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 loading image, or for peak characteristics,. Default is the matplotlib gray colormap (*plt.cm.gray*).

- **no_nans** *(bool)* – If True, removes NaN's from the loading plots.

- **per_row** *(int)* – The number of plots in each row, when the *same_window* parameter is True.

- **axes_decor** *(str or None, optional)* – One of: 'all', 'ticks', 'off', or None Controls how the axes are displayed on each image; default is 'all'. If 'all', both ticks and axis labels will be shown. If 'ticks', no axis labels will be shown, but ticks/labels will. If 'off', all decorations and frame will be disabled. If None, no axis decorations will be shown, but ticks/frame will

See also:

- **plot_bss_factors()**, **plot_bss_results()**

**plot_bss_results** *(factors_navigator='smart_auto', loadings_navigator='smart_auto', factors_dim=2, loadings_dim=2)*

Plot the blind source separation factors and loadings.

Unlike **plot_bss_factors()** and **plot_bss_loadings()**, this method displays one component at a time. Therefore it provides a more compact visualization than then other two methods. The loadings and factors are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

**Parameters**

- **factors_navigator** *(str, None, or BaseSignal (or subclass))* – One of: 'smart_auto', 'auto', None, 'spectrum' or a BaseSignal object. 'smart_auto' (default) displays sliders if the navigation dimension is less than 3. For a description of the other options see the **plot()** documentation for details.

- **loadings_navigator** *(str, None, or BaseSignal (or subclass))* – See the factors_navigator parameter
• **factors_dim** *(int)* – Currently HyperSpy cannot plot a signal when the signal dimension is higher than two. Therefore, to visualize the BSS results when the factors or the loadings have signal dimension greater than 2, the data can be viewed as spectra (or images) by setting this parameter to 1 (or 2). (The default is 2)

• **loadings_dim** *(int)* – See the **factors_dim** parameter

See also:

plot_bss_factors(), plot_bss_loadings(), plot_decomposition_results()

plot_cluster_distances*(cluster_ids=None, calibrate=True, same_window=True, with_centers=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', title=None, **kwargs)*

Plot the euclidian distances to the centroid of each cluster.

In case of 1D navigation axis, each line can be toggled on and off by clicking on the legended line.

**Parameters**

• **cluster_ids** *(None, int, or list of ints)* – if None (default), returns maps of all components using the number_of_cluster was defined when executing cluster. Otherwise it raises a ValueError. if int, returns maps of cluster labels with ids from 0 to given int. if list of ints, returns maps of cluster labels with ids in given list.

• **calibrate** *(bool)* – if True, calibrates plots where calibration is available from the axes_manager. If False, plots are in pixels/channels.

• **same_window** *(bool)* – if True, plots each factor to the same window. They are not scaled. Default is True.

• **title** *(string)* – Title of the plot.

• **with_centers** *(bool)* – If True, also returns figure(s) with the cluster centers for the given cluster_ids.

• **cmap** *(matplotlib 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** *(['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

See also:

plot_cluster_signals(), plot_cluster_results(), plot_cluster_labels()

plot_cluster_labels*(cluster_ids=None, calibrate=True, same_window=True, with_centers=False, cmap=<matplotlib.colors.LinearSegmentedColormap object>, no_nans=False, per_row=3, axes_decor='all', title=None, **kwargs)*

Plot cluster labels from a cluster analysis. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on the legended line.

**Parameters**
HyperSpy Documentation, Release 1.6.0

• `cluster_ids` (*None, int, or list of ints*) – if None (default), returns maps of all components using the number of cluster was defined when executing `cluster`. Otherwise it raises a ValueError. if int, returns maps of cluster labels with ids from 0 to given int. if list of ints, returns maps of cluster labels with ids in given list.

• `calibrate` (*bool*) – if True, calibrates plots where calibration is available from the `axes_manager`. If False, plots are in pixels/channels.

• `same_window` (*bool*) – if True, plots each factor to the same window. They are not scaled. Default is True.

• `title` (*string*) – Title of the plot.

• `with_centers` (*bool*) – If True, also returns figure(s) with the cluster centers for the given `cluster_ids`.

• `cmap` (*matplotlib 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` (*{'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.

See also:

`plot_cluster_signals()`, `plot_cluster_results()`.

`plot_cluster_results` (*centers_navigator='smart_auto', labels_navigator='smart_auto', centers_dim=2, labels_dim=2*)

Plot the cluster labels and centers.

Unlike `plot_cluster_labels` and `plot_cluster_signals`, this method displays one component at a time. Therefore it provides a more compact visualization than then other two methods. The labels and centers are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

Parameters

• `labels_navigator` (*centers_navigator*) –

• `None, "spectrum", Signal` (*"auto"*) – “smart_auto” (default) displays sliders if the navigation dimension is less than 3. For a description of the other options see `plot` documentation for details.

• `centers_dims` (*labels_dim*) – Currently HyperSpy cannot plot signals of dimension higher than two. Therefore, to visualize the clustering results when the centers or the labels have signal dimension greater than 2 we can view the data as spectra(images) by setting this parameter to 1(2). (Default 2)

See also:

`plot_cluster_signals()`, `plot_cluster_labels()`.

`plot_cluster_signals` (*signal='mean', cluster_ids=None, calibrate=True, same_window=True, comp_label='Cluster centers', per_row=3*)

Plot centers from a cluster analysis.
Parameters

- **signal** ("mean", "sum", "centroid", optional) – If “mean” or “sum” return the mean signal or sum respectively over each cluster. If “centroid”, returns the signals closest to the centroid.

- **cluster_ids** (None, int, or list of ints) – If None, returns maps of all clusters. If int, returns maps of clusters with ids from 0 to given int. If list of ints, returns maps of clusters with ids in given list.

- **calibrate** – 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 center to the same window. They are not scaled.

- **comp_label** (string) – 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)

- **per_row** (int) – the number of plots in each row, when the same_window parameter is True.

See also:

*plot_cluster_labels()*

*plot_decomposition_factors*(comp_ids=None, calibrate=True, same_window=True, title=None, cmap=plt.cm.gray, per_row=3, **kwargs)

Plot factors from a decomposition. In case of 1D signal axis, each factors line can be toggled on and off by clicking on their corresponding line in the legend.

Parameters

- **comp_ids** (None, int, or list of 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 images, or for peak characteristics. Default is the matplotlib gray colormap (plt.cm.gray).

- **per_row** (int) – The number of plots in each row, when the same_window parameter is True.

See also:

*plot_decomposition_loadings(), plot_decomposition_results()*

*plot_decomposition_loadings*(comp_ids=None, calibrate=True, same_window=True, title=None, with_factors=False, cmap=plt.cm.gray, per_row=3, axes_decor='all', **kwargs)
Plot loadings from a decomposition. In case of 1D navigation axis, each loading line can be toggled on and off by clicking on the legended line.

**Parameters**

- **comp_ids (None, int, or list (of 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 loadings images, or for peak characteristics. Default is the matplotlib gray colormap (`plt.cm.gray`).

- **no_nans (bool)** – If `True`, removes `NaN`'s from the loading plots.

- **per_row (int)** – The number of plots in each row, when the `same_window` parameter is `True`.

- **axes_decor (str or None, optional)** – One of: 'all', 'ticks', 'off', or `None`. Controls how the axes are displayed on each image; default is 'all'. If 'all', both ticks and axis labels will be shown. If 'ticks', no axis labels will be shown, but ticks/labels will. If 'off', all decorations and frame will be disabled. If `None`, no axis decorations will be shown, but ticks/frame will.

See also:

- `plot_decomposition_factors()` , `plot_decomposition_results()`

**plot_decomposition_results** (factors_navigator='smart_auto', loadings_navigator='smart_auto', factors_dim=2, loadings_dim=2)

Plot the decomposition factors and loadings.

Unlike `plot_decomposition_factors()` and `plot_decomposition_loadings()`, this method displays one component at a time. Therefore it provides a more compact visualization than the other two methods. The loadings and factors are displayed in different windows and each has its own navigator/sliders to navigate them if they are multidimensional. The component index axis is synchronized between the two.

**Parameters**

- **factors_navigator (str, None, or BaseSignal (or subclass))** – One of: 'smart_auto', 'auto', `None`, 'spectrum' or a `BaseSignal` object. 'smart_auto' (default) displays sliders if the navigation dimension is less than 3. For a description of the other options see the `plot()` documentation for details.

- **loadings_navigator (str, None, or BaseSignal (or subclass))** – See the `factors_navigator` parameter
• **factors_dim** *(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

    _add_dictionary(dictionary=None)

    _check_name(name, existing=False)

    _save(name, dictionary)

    _set_nice_description(node, names)

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()
**class hyperspy.signal.SpecialSlicersSignal**(obj, isNavigation)

**Bases:** hyperspy.misc.slicing.SpecialSlicers

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)>
```

```python
data
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```python
axes_manager[0].scale = 0.5
```

```python
axes_manager[0].axis
array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5])
```

```python
isig[0.5:4].data
array([1, 2, 3, 4, 5, 6, 7])
```

```python
isig[0.5:4].data
array([1, 2, 3])
```

```python
isig[0.5:4:2].data
array([1, 3])
```

```python
axes_manager[0].units = 'µm'
```

```python
isig[:'2000 nm'].data
array([0, 1, 2, 3])
```

**hyperspy.signal._change_API_comp_label**(title, comp_label)

**hyperspy.signal._plot_x_results**(factors, loadings, factors_navigator, loadings_navigator, factors_dim, loadings_dim)

**hyperspy.signal_tools module**

**class hyperspy.signal_tools.BackgroundRemoval**(signal, background_type='Power law', polynomial_order=2, fast=True, plot_remainder=True, zero_fill=False, show_progressbar=None, model=None)

**Bases:** hyperspy.signal_tools.SpanSelectorInSignal1D

```python
_background_type_changed(old, new)
```

```python
_fast_changed(old, new)
```

```python
_fit()
```

```python
_polynomial_order_changed(old, new)
```

```python
_ss_left_value_changed(old, new)
```

```python
_ss_right_value_changed(old, new)
```

```python
_update_line()
```

```python
apply()
```

```python
bg_to_plot(axes_manager=None, fill_with=nan)
```

```python
create_background_line()
```

```python
create_remainder_line()
```

```python
disconnect()
```

```python
gui(display=True, toolkit=None, **kwargs)
```

Display or return interactive GUI element if available.

---

**3.1. hyperspy package**
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()
```

```python
class hyperspy.signal_tools.ButterworthFilter (signal)
Bases: hyperspy.signal_tools.Smoothing
```

```python
_cutoff_frequency_ratio_changed (old, new)
_order_changed (old, new)
_type_changed (old, new)
apply()
```

```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
model2plot (axes_manager=None)
```

```python
class hyperspy.signal_tools.DerivativeTextHandler
Bases: object
```

```python
legend_artist (legend, orig_handle, fontsize, handlebox)
```

```python
class hyperspy.signal_tools.DerivativeTextParameters (text, color)
Bases: object
```

```python
class hyperspy.signal_tools.EdgesRange (signal, active=None)
Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
```

```python
_clear_markers (edges=None)
_get_edges_info_within_energy_axis()
_keep_valid_edges()
_on_figure_changed()
_plot_labels (active=None, complementary=None)
```

```python
check_btn_state()
```
gui \( (display=True, toolkit=None, **kwargs) \)
Display or return interactive GUI element if available.

Parameters

- **display** \( (\text{bool}) \) – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.

- **toolkit** \( (\text{str, iterable of strings or None}) \) – 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.

---

3.1. hyperSpy package

```python
class hyperSpy.signal_tools.ImageContrastEditor(image)
Bases: traits.has_traits.HasTraits

_on_complementary()
_update_active_edge(change)
_update_table()

apply()
close()
create_axis()

gui \( (display=True, toolkit=None, **kwargs) \)
Display or return interactive GUI element if available.

Parameters

- **display** \( (\text{bool}) \) – 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
mpl_help = 'See the matplotlib SymLogNorm for more information.'
```

```python
plot_histogram(*max_num_bins=250*)
Plot a histogram of the data.

**Parameters**
`max_num_bins` (*int*, default 250) – When estimating the bins using one of the str methods, the number of bins is capped by this number to avoid a MemoryError being raised by `numpy.histogram()`.

**Returns**
```
Return type None
```

```python
reset()```

```python
span_selector_switch(on)
```

```python
update_histogram()
```

```python
update_line()
```

```python
update_span_selector(*args, **kwargs)
```

```python
update_span_selector_traits(*args, **kwargs)
```

```python
class hyperspy.signal_tools.IntegrateArea(*signal, signal_range=None*)
Bases: hyperspy.signal_tools.SpanSelectorInSignal1D
```

```python
apply()
```

```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
class hyperspy.signal_tools.LineInSignal1D(*signal*)
Bases: traits.has_traits.HasTraits
```

```python
Adds a vertical draggable line to a spectrum that reports its position to the position attribute of the class.
```

```python
position
The position of the vertical line in the one dimensional signal. Moving the line changes the position but the reverse is not true.
```

**Type** float

```python
on
Turns on and off the line
```

**Type** bool

```python
color
The color of the line. It automatically redraws the line.
```
Type  wx.Colour

_,color_changed_(old, new)

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.PeaksFinder2D(signal, method, peaks=None, **kwargs)
Bases: traits.has_traits.HasTraits

_,find_peaks_current_index_(method)

_,get_parameters_(method)

_,method_changed_(old, new)

_,normalise_method_name_(method)

_,parameter_changed_(old, new)

_,parse_paramaters_initial_values_(*kwargs)

_,peaks_to_marker_(markersize=20, add_numbers=True, color='red')

_,plot_markers()

_,set_parameters_observer()

_,update_peak_finding_(method=None)

close()

compute_navigation()

disconnect()

#,display=True, toolkit=toolkit, **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.

#,random_navigation_position()

gxc_template = None

class hyperspy.signal_tools.Signal1DCalibration(signal)
Bases: hyperspy.signal_tools.SpanSelectorInSignal1D

_,left_value_changed_(old, new)

_,right_value_changed_(old, new)

_,update_calibration_(*args, **kwargs)

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

**_differential_order_changed** *(old, new)*

**_line_color_changed** *(old, new)*

**_line_color_ipy_changed** *

**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

**_number_of_iterations_changed** *(old, new)*

**_smoothing_parameter_changed** *(old, new)*

**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)
```

```python
class hyperspy.signal_tools.SmoothingSavitzkyGolay (signal)
Bases: hyperspy.signal_tools.Smoothing
```

```python
    _decrease_window_length_fired ()
    _differential_order_changed (old, new)
    _increase_window_length_fired ()
    _polynomial_order_changed (old, new)
    _window_length_changed (old, new)
```

```python
apply ()
```

```python
diff_model2plot (axes_manager=None)
```

```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
model2plot (axes_manager=None)
```

```python
class hyperspy.signal_tools.SmoothingTV (signal)
Bases: hyperspy.signal_tools.Smoothing
```

```python
    _smoothing_parameter_changed (old, new)
```

```python
apply ()
```

```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.
HyperSpy Documentation, Release 1.6.0

class hyperspy.signal_tools.SpanSelectorInSignal1D(signal)
    Bases: traits.has_traits.HasTraits
    property is_span_selector_valid
    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
    _add_noise_changed(old, new)
    _click_to_show_instructions_fired()
    _index_changed(old, new)
    _interpolator_kind_changed(old, new)
    _reset_line()
    _show_derivative_histogram_fired()
    _spline_order_changed(old, new)
    _ss_left_value_changed(old, new)
    _ss_right_value_changed(old, new)
    _threshold_changed(old, new)
    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 interable 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()**

```python
hyperspy.signal_tools._get_background_estimator(background_type, polynomial_order=1)
```

Assign 1D component to specified background type.

**Parameters**

- `background_type` *(str)* – The name of the component to model the background.
- `polynomial_order` *(int, optional)* – The polynomial order used in the polynomial component

**Raises** `ValueError` – When the background type is not a valid string.

**Returns**

- `background_estimator` *(Component1D)* – The component modeling the background.
- `bg_line_range` *(‘full’ or ‘from_left_range’)* – The range to draw the component (used in the Background Removal tool)

---

**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._toolkits_to_string(toolkits)

hyperspy.ui_registry.add_gui_method(toolkey)

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.
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, 666
hyperspy._components, 291
hyperspy._components.arctan, 237
hyperspy._components.bleasdale, 239
hyperspy._components.doniach, 240
hyperspy._components.eels_arctan, 243
hyperspy._components.eels_cl_edge, 246
hyperspy._components.eels_double_power_law, 248
hyperspy._components.eels_vignetting, 250
hyperspy._components.error_function, 251
hyperspy._components.exponential, 252
hyperspy._components.expression, 254
hyperspy._components.gaussian, 256
hyperspy._components.gaussian2d, 258
hyperspy._components.gaussianhf, 261
hyperspy._components.heaviside, 263
hyperspy._components.logistic, 265
hyperspy._components.lorentzian, 267
hyperspy._components.offset, 270
hyperspy._components.pes_core_line_shape, 270
hyperspy._components.pes_see, 271
hyperspy._components.pes_voigt, 271
hyperspy._components.polynomial, 274
hyperspy._components.polynomial_deprecate, 276
hyperspy._components.power_law, 277
hyperspy._components.rc, 279
hyperspy._components.scalable_fixed_pattern, 280
hyperspy._components.skew_normal, 282
hyperspy._components.split_pvoigt, 285
hyperspy._components.voigt, 286
hyperspy._components.volume_plasmon_drude, 289
hyperspy._lazy_signals, 558
hyperspy._signals, 366
hyperspy._signals.common_signal1d, 291
hyperspy._signals.common_signal2d, 292
hyperspy._signals.complex_signal, 292
hyperspy._signals.complex_signal1d, 296
hyperspy._signals.complex_signal2d, 297
hyperspy._signals.dielectric_function, 300
hyperspy._signals.eds, 302
hyperspy._signals.eds_sem, 312
hyperspy._signals.eds_tem, 314
hyperspy._signals.eels, 321
hyperspy._signals.hologram_image, 331
hyperspy._signals.lazy, 336
hyperspy._signals.signal1d, 345
hyperspy._signals.signal2d, 358
hyperspy.api, 558
hyperspy.api_nogui, 559
hyperspy.axes, 560
hyperspy.component, 567
hyperspy.components1d, 574
hyperspy.components2d, 574
hyperspy.datasets, 369
hyperspy.datasets.artificial_data, 366
hyperspy.datasets.example_signals, 369
hyperspy.decorators, 574
hyperspy.defaults_parser, 575
hyperspy.docstrings, 370
hyperspy.docstrings.model, 370
hyperspy.docstrings.parameters, 370
hyperspy.docstrings.plot, 370
hyperspy.docstrings.signal, 370
hyperspy.docstrings.signal1d, 370
hyperspy.docstrings.signal2d, 370
hyperspy.drawing, 405
hyperspy.drawing._markers, 376
hyperspy.drawing._markers.horizontal_line, 370
hyperspy.drawing._markers.horizontal_line_segment, 371
hyperspy.drawing._markers.line_segment, 372
hyperspy.drawing._markers.point, 373
hyperspy.drawing._markers.rectangle, 374
hyperspy.drawing._markers.text, 374
hyperspy.drawing._markers.vertical_line,
### Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Component__axes_manager</td>
<td>(hyper-spy.component.Component attribute), 567</td>
<td></td>
</tr>
<tr>
<td>_F()</td>
<td>(in module hyperspy.misc.eels.electron_inelastic_mean_free_path), 476</td>
<td></td>
</tr>
<tr>
<td>_MS_parse()</td>
<td>(hyper-spy.io_plugins.sur.DigitalSurfHandler method), 441</td>
<td></td>
</tr>
<tr>
<td>_Parameter__ext_bounded</td>
<td>(hyper-spy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Parameter__ext_force_positive</td>
<td>(hyper-spy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Parameter__free</td>
<td>(hyperspy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Parameter__number_of_elements</td>
<td>(hyperspy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Parameter__twin</td>
<td>(hyperspy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Parameter__value</td>
<td>(hyperspy.component.Parameter attribute), 571</td>
<td></td>
</tr>
<tr>
<td>_Samfire__active_strategy_ind</td>
<td>(hyper-spy.samfire.Samfire attribute), 610</td>
<td></td>
</tr>
<tr>
<td>_get_ipr_header()</td>
<td>(in module hyperspy.io_plugins.edax), 413</td>
<td></td>
</tr>
<tr>
<td>_get_spc_header()</td>
<td>(in module hyperspy.io_plugins.edax), 413</td>
<td></td>
</tr>
<tr>
<td>_active</td>
<td>(hyperspy.component.Component attribute), 567</td>
<td></td>
</tr>
<tr>
<td>_active_background_components()</td>
<td>(hyper-spy.models.edsmode.EDSModel property), 517</td>
<td></td>
</tr>
<tr>
<td>_add_background_windows_markers()</td>
<td>(hyper-spy_signals.eds.EDS_mixin method), 302</td>
<td></td>
</tr>
<tr>
<td>_add_colorbar()</td>
<td>(hyper-spy.drawing.image.ImagePlot method), 387</td>
<td></td>
</tr>
<tr>
<td>_add_dictionary()</td>
<td>(hyper-spy.signal.ModelManager method), 656</td>
<td></td>
</tr>
<tr>
<td>_add_edges_from_subshells_names()</td>
<td>(hyperspy.models.eelsmode.EELSMoodel method), 524</td>
<td></td>
</tr>
<tr>
<td>_add_noise_changed()</td>
<td>(hyper-spy.signal_tools.SpikesRemoval method), 664</td>
<td></td>
</tr>
<tr>
<td>_add_patch_to()</td>
<td>(hyper-spy.drawing.widgets.range.RangeWidget method), 382</td>
<td></td>
</tr>
<tr>
<td>_add_patch_to()</td>
<td>(hyper-spy.drawing.widget.ResizersMixin method), 402</td>
<td></td>
</tr>
<tr>
<td>_add_patch_to()</td>
<td>(hyper-spy.drawing.widget.WidgetBase method), 404</td>
<td></td>
</tr>
<tr>
<td>_add_signal_range_in_pixels()</td>
<td>(hyper-spy.models.model1d.Model1D method), 535</td>
<td></td>
</tr>
<tr>
<td>_add_signal_range_in_pixels()</td>
<td>(hyper-spy.models.model2d.Model2D method), 540</td>
<td></td>
</tr>
<tr>
<td>_add_single()</td>
<td>(hyperspy.events.EventSuppressor method), 578</td>
<td></td>
</tr>
<tr>
<td>_add_spc_metadata()</td>
<td>(in module hyperspy.io_plugins.edax), 413</td>
<td></td>
</tr>
<tr>
<td>_add_vertical_lines_groups()</td>
<td>(hyper-spy_signals.eds.EDS_mixin method), 302</td>
<td></td>
</tr>
<tr>
<td>_additional_slicing_targets</td>
<td>(hyper-spy.signal.BaseSignal attribute), 613</td>
<td></td>
</tr>
<tr>
<td>_alias_signal_types</td>
<td>(hyper-spy_signals.dielectric_function.DielectricFunction_mixin attribute), 301</td>
<td></td>
</tr>
<tr>
<td>_alias_signal_types</td>
<td>(hyper-spy_signals.eels.EELSSpectrum_mixin attribute), 321</td>
<td></td>
</tr>
</tbody>
</table>

---

INDEX
_freedman_bw_dask() (in module hyperspy.misc.hist_tools), 492
_function() (hyperspy._components.offset.Offset method), 270
_function() (hyperspy._components.polynomial_deprecated.Polynomial method), 276
_function() (hyperspy._components.split_pvoigt.SplitVoigt method), 285
_function4odr() (hyperspy.models.model1d.Model1D method), 536
_function4odr() (hyperspy.models.model2d.Model2D method), 540
_gamma_changed() (hyperspy.signal_tools.ImageContrastEditor method), 659
_get_E0() (hyperspy_components.eels_cl_edge.EELSCLEdge method), 247
_get_EELS_exposure_time() (hyperspy.io_plugins.digital_micrograph.ImageObject method), 412
_get_active() (hyperspy.component.Component method), 567
_get_amplitude() (hyperspy_signals.complex_signal.ComplexSignal method), 292
_get_amplitude() (hyperspy_signals.complex_signal.ComplexSignal mixin method), 293
_get_amplitude() (hyperspy_signals.complex_signal.LazyComplexSignal method), 295
_get_array_slices() (hyperspyAxes.DataAxis method), 564
_get_array_slices() (hyperspy.misc.slicing.FancySlicing method), 503
_get_axes() (hyperspy.drawing.widget.WidgetBase method), 404
_get_axes() (hyperspy.io_plugins.protochips.ProtochipsCSV method), 436
_get_axes_dicts() (hyperspyAxes.AxesManager method), 561
_get_axes_in_natural_order() (hyperspy_axes.AxesManager method), 561
_get_axis_attribute_values() (hyperspy_axes.AxesManager method), 561
_get_background_estimator() (in module hyperspy.signal_tools), 665
_get_bbox_from_textbox_patch() (hyperspy.misc.label_position.SpectrumLabelPosition method), 495
_get_beam_energy() (hyperspy_signals eds. EDS mixin method), 302
_get_bmax() (hyperspy.component.Parameter method), 571
_get_bmin() (hyperspy.component.Parameter method), 571
_get_bytes() (hyperspy.io_plugins.sur.DigitalSurfHandler method), 443
_get_cluster_algorithm() (hyperspy.learn.mva.MVA method), 448
_get_cluster_preprocessing_algorithm() (hyperspy.learn.mva.MVA method), 448
_get_cluster_signal() (hyperspy.learn.mva.MVA method), 449
_get_cluster_signals_factors() (hyperspy.signal.MVATools method), 644
_get_collection_angle() (hyperspy_components.eels_cl_edge.EELSCLEdge method), 247
_get_current_range() (hyperspy.signal_tools.ImageContrastEditor method), 659
_get_current_signal_index() (hyperspy.misc.label_position.SpectrumLabelPosition method), 495
_get_dask_chunks() (hyperspy_signals.lazy.LazySignal method), 337
_get_data() (hyperspy.drawing.signal1d.Signal1DLine method), 391
_get_data() (hyperspy.signal_tools.ImageContrastEditor method), 659
_get_data_array() (hyperspy.io_plugins.digital_micrograph.ImageObject method), 412
_get_data_shape() (hyperspy.drawing.marker.MarkerBase method), 388
_get_data_slice() (hyperspyAxes.AxesManager method), 561
_get_date() (hyperspy.io_plugins.digital_micrograph.ImageObject method), 412
_get_date_time() (in module hyperspy.io_plugins.fei), 426
_get_datetime() (hyper¬spy.io_plugins.phenom.ElidReader method), 435

_get_derivative() (in module hyper¬spy.learn.mva), 460

_get_detector_metadata_dict() (in module hyperspy.io_plugins.emd), 425

_get_diff() (hyper¬spy.drawing._widgets.line2d.Line2DWidget method), 379

_get_detector_str() (hyper¬spy.axes.AxesManager method), 561

_get_dimensions() (hyper¬spy.io_plugins.emd.FeiEMDReader method), 424

_get_distance_array() (hyper¬spy.samfire_utils.strategy.LocalStrategy method), 547

_get_dm_kwargs_extratag() (in module hyper¬spy.io_plugins.tiff), 444

_get_dose() (hyper¬spy_signals.eds_tem.EDSTEM_mixin method), 314

_get_edges() (hyper¬spy_signals.eels.EELSSpectrum_mixin method), 321

_get_edges_info_within_energy_axis() (hyper¬spy.signal_tools.EdgesRange method), 658

_get_edges_to_plot() (hyper¬spy_signals.eels.EELSSpectrum_mixin method), 321

_get_element_and_line() (in module hyper¬spy.misc.eeds.utils), 469

_get_emd_group_type() (hyper¬spy.io_plugins.emd.EMD_NCEM static method), 423

_get_energy_xray_line() (in module hyper¬spy.misc.eeds.utils), 469

_get_factors() (hyper¬spy.signal.MVATools method), 644

_get_fine_structure_active() (hyper¬spy_components.eels_cl_edge.EELSCLEdge method), 247

_get_fine_structure_width() (hyper¬spy_components.eels_cl_edge.EELSCLEdge method), 247

_get_first_ionization_edge_energy() (hyper¬spy.models.eelsmodel.EELSModel method), 524

_get_float() (hyper¬spy.io_plugins.sur.DigitalSurfHandler method), 443

_get_free() (hyper¬spy.component.Parameter method), 571

_get_func_from_pos() (hyper¬spy.drawing._widgets.line2d.Line2DWidget method), 379

_get_histogram() (hyper¬spy.signal_tools.ImageContrastEditor method), 659

_get_html_print_items() (hyper¬spy.misc.utils.DictionaryTreeBrowser method), 510

_get_imag() (hyper¬signals.complex_signal.ComplexSignal_mixin method), 293

_get_imagej_kwargs() (in module hyper¬spy.io_plugins.tif), 444

_get_index() (hyper¬spy.axes.DataAxis method), 564

_get_index_from_value_with_units() (hyper¬spy.axes.UnitConversion method), 566

_get_indices() (hyper¬spy.drawing.widget.DraggableWidgetBase method), 399

_get_int16() (hyper¬spy.io_plugins.sur.DigitalSurfHandler method), 443

_get_int32() (hyper¬spy.io_plugins.sur.DigitalSurfHandler method), 443

_get_keys_from_group() (in module hyper¬spy.io_plugins.emd), 425

_get_label() (hyper¬spy.io_plugins.semper_unf.SemperFormat method), 440

_get_line() (hyper¬spy.signal_tools.ImageContrastEditor method), 659

_get_line_energy() (hyper¬spy_signals.eeds.EDS_mixin method), 302

_get_line_normal() (hyper¬spy.drawing._widgets.line2d.Line2DWidget method), 379

_get_lines_from_elements() (hyper¬spy_signals.eeds.EDS_mixin method), 302

_get_loadings() (hyper¬spy.signal.MVATools method), 644

_get_local_time_zone() (hyper¬spy.io_plugins.emd.FeiEMDReader method), 424

_get_local_time_zone() (hyper¬spy.io_plugins.phenom.ElidReader method), 435

_get_long_description() (hyper¬spy.component.Component method), 567

_get_mapping() (hyper¬spy.io_plugins.emd.FeiEMDReader method),
_get_mapping()

_get_metadata()

_get_metadata_time_axis()

_get_microscope_name()

_get_name()

_get_navigation_axes_dicts()

_get_navigation_signal()

_get_navigation_sliders()

_get_number_of_components_for_clustering()

_get_offset()

_get_original_metadata()

_get_parameters()

_get_patch_bounds()

_get_patch_xy()
_get_resizer_pos() (hyperSpy.drawing.widget.ResizersMixin method), 402

_get_resizer_size() (hyperSpy.drawing.widget.ResizersMixin method), 402

_get_scale() (in module hyperspy.models.edsmode1), 522

_get_scale_unit() (in module hyperspy.io_plugins.tiff), 444

_get_scales_from_x_y_resolution() (in module hyperspy.io_plugins.tiff), 444

_get_shapes() (in module hyperspy.misc.signal_tools), 503

_get_short_description() (hyperSpy.component.Component method), 567

_get_sigma() (in module hyperspy.models.edsmode1), 522

_get_signal_axes_dicts() (hyperSpy.axes.AxesManager method), 561

_get_signal_signal() (hyperSpy.signal.BaseSignal method), 614

_get_simplified_mode() (in module hyperspy.io_plugins.fei), 426

_get_size() (hyperSpy.drawing.widget.ResizableDraggableWidgetBase method), 401

_get_span_width() (hyperSpy.drawing._widgets.range.ModifiableSpanSelector method), 381

_get_span_x() (hyperSpy.drawing._widgets.range.ModifiableSpanSelector method), 381

_get_start_energy() (hyperSpy.models.eelsmodel.EELSModel method), 524

_get_str() (hyperSpy.io_plugins.sur.DigitalSurfHandler method), 443

_get_string() (hyperSpy.drawing._widgets.label.LabelWidget method), 378

_get_tags_dict() (in module hyperspy.io_plugins.tiff), 444

_get_textbox_pos() (hyperSpy.misc.label_position.SpectrumLabelPosition method), 495

_get_time() (hyperSpy.io_plugins.digital_micrograph.ImageObject method), 412

_get_twin() (hyperSpy.component.Parameter method), 571

_get_uint32() (hyperSpy.io_plugins.sur.DigitalSurfHandler method), 443

_get_undefined_axes_list() (hyperSpy.signal.BaseSignal method), 614

_get_unit() (hyperSpy.io_plugins.phenom.ElidReader method), 435

_get_unit_factor() (hyperSpy.io_plugins.phenom.ElidReader method), 435

_get_value() (hyperspy.component.Parameter method), 571

_get_value_with_unit() (hyperSpy.io_plugins.phenom.ElidReader method), 435

_get_variance() (hyperspy.model.BaseModel method), 588

_get_vertex() (hyperSpy.roi.BaseInteractiveROI method), 599

_get_widget_type() (hyperSpy.roi.CircleROI method), 601

_get_widget_type() (hyperSpy.roi.Line2DROI method), 602

_get_widget_type() (hyperSpy.roi.Point1DROI method), 605

_get_widget_type() (hyperSpy.roi.Point2DROI method), 606

_get_widget_type() (hyperSpy.roi.RectangularROI method), 607

_get_width_indicator_coords() (hyperSpy.drawing._widgets.line2d.Line2DWidget method), 379

_get_work_dict_key_value() (hyperSpy.io_plugins.sur.DigitalSurfHandler method), 443

_get_xray_lines() (hyperSpy.signals.eds.EDS_mixin method), 303

_get_xray_lines_family() (in module hyperspy.misc.eds.utils), 469

_get_xray_lines_in_spectral_range() (hyperSpy.signals.eds.EDS_mixin method), 303

_getlink() (in module hyperspy.io_plugins.nexus), 430

_gradient_huber() (hyperSpy.models.model1d.Model1D method), 536

_gradient_huber() (hyperSpy.models.model2d.Model2D method), 536

Index 687
_gradient_ls() (hyper-spy.models.model1d.Model1D method), 536
_is_linear_axis() (in module hyperspy.io_plugins.nexus), 430
_is_marker_static() (hyper-spy.drawing.marker.MarkerBase method), 388
_is_navigation_multidimensional() (hyper-spy.component.Component property), 567
_is_numeric_data() (in module hyperspy.io_plugins.nexus), 430
_is_prismatic_file() (hyper-spy.io_plugins.emd.EMD_NCEM property), 423
_is_protochips_csv_file() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_is_py4DSTEM_file() (hyper-spy.io_plugins.emd.EMD_NCEM property), 423
_is_target() (hyperspy.events.EventSuppressor method), 578
_is_tuple_target() (hyperspy.events.EventSuppressor method), 578
_iter_join() (in module hyperspy.misc.model_tools), 501
_iter_read_chunks() (hyper-spy.io_plugins.bruker.SFSTreeItem method), 408
_iter_read_compr_chunks() (hyper-spy.io_plugins.bruker.SFSTreeItem method), 408
_iterate_signal() (hyperspy._signals.lazy.LazySignal method), 337
_iterate_signal() (hyperspy.signal.BaseSignal method), 614
_iterate_signal() (hyperspy._signals.signal1d.LazySignal1D attribute), 345
_iterate_signal() (hyperspy._signals.signal2d.LazySignal2D attribute), 358

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spy.signal_tools.PeaksFinder2D._model2plot()</td>
<td>661</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._model_function()</td>
<td>659</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._mountains_object_types</td>
<td>443</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._normalize_components()</td>
<td>461</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._number_of_elements()</td>
<td>571</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._number_of_iterations_changed()</td>
<td>562</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_blit_draw()</td>
<td>386</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_close()</td>
<td>386</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_close()</td>
<td>390</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_index_changed()</td>
<td>658</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_index_changed()</td>
<td>561</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_navigate()</td>
<td>399</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_offset_changed()</td>
<td>389</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_position_widget_close()</td>
<td>561</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_scale_changed()</td>
<td>536</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_size_changed()</td>
<td>562</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_twin_update()</td>
<td>379</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_twin_update()</td>
<td>404</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_change()</td>
<td>599</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_moved()</td>
<td>536</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_moved()</td>
<td>377</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_moved()</td>
<td>377</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_moved()</td>
<td>377</td>
</tr>
<tr>
<td>spy.signal_tools.PeaksFinder2D._on_widget_moved()</td>
<td>377</td>
</tr>
</tbody>
</table>
HyperSpy Documentation, Release 1.6.0

_onmousemove() (hyper-spy.drawing._widgets.rectangles.SquareWidget method), 384
_onmousemove() (hyper-spy.drawing._widgets.vertical_line.VerticalLineWidget method), 386
_onmousemove() (hyper-spy.drawing.widget.DraggableWidgetBase method), 399
_onset_energy() (hyper-spy_components.eels_cl_edge.EELSCLEdge method), 247
_order_changed() (hyper-spy.signal_tools.ButterworthFilter method), 658
_pack_comment() (hyper-spy.io_plugins.sur.DigitalSurfHandler method), 443
_pack_data() (hyper-spy.io_plugins.sur.DigitalSurfHandler method), 443
_pack_private() (hyper-spy.io_plugins.sur.DigitalSurfHandler method), 443
_package_values() (hyper-spy.samfire_utils.strategy.GlobalStrategy method), 547
_parameter_changed() (hyper-spy.signal_tools.PeaksFinder2D method), 661
_parse_attribute() (hyper-spy.io_plugins.emd.EMD_NCEM method), 423
_parse_axes() (hyperspy.roi.BaseROI method), 600
_parse_axis() (hyper-spy.io_plugins.emd.EMD_NCEM static method), 423
_parse_beam_current_FEI() (in module hyperspy.io_plugins.tifff), 444
_parse_bounds_args() (hyper-spydrawing._widgets.range.RangeWidget method), 382
_parse_bounds_args() (hyper-spydrawing._widgets.rectangles.RectangleWidget method), 383
_parse_calibration_file_name() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_calibration_filename() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_detector_name() (in module hyperspy.io_plugins.emd), 425
_parse_frame_time() (hyper-spy.io_plugins.emd.FeiEMDReader method), 424
_parse_from_file() (in module hyperspy.io_plugins.nexus), 431
_parse_header() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_image() (hyper-spy.io_plugins.brucker.HyperHeader method), 406
_parse_image_display() (hyper-spy.io_plugins.emd.FeiEMDReader method), 424
_parse_metadata() (hyper-spy.io_plugins.emd.EMD_NCEM method), 423
_parse_metadata() (in module hyperspy.io_plugins.emd), 425
_parse_metadata_group() (hyper-spy.io_plugins.emd.FeiEMDReader method), 424
_parse_metadata_header() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_notes() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_only_lines() (in module hyperspy.misc.eds.utils), 469
_parse_original_metadata() (hyper-spy.io_plugins.emd.EMD_NCEM method), 423
_parse_parameters_initial_values() (hyperspy.signal_tools.PeaksFinder2D method), 661
_parse_quantity() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_quantity_units() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_sb_position() (in module hyperspy_signals.hologram_image), 336
_parse_sb_size() (in module hyperspy_signals.hologram_image), 336
_parse_scale_unit() (in module hyperspy.io_plugins.tifff), 444
_parse_string() (hyper-spy.io_plugins.protochips.ProtochipsCSV method), 436
_parse_string() (hyper-spy.io_plugins.digital_micrograph.ImageObject method), 412
_parse_string() (in module hyperspy.io_plugins.tifff), 444
_parse_string_for_slice() (hyper-
_read_oxide() (hyper-spy.io_plugins.phenom.ElidReader method), 436

_remove_axis() (hyperspy.signal.BaseSignal method), 616

_remove_background_cli() (hyperspy._signals.signal1d.Signal1D method), 346

_remove_one_axis() (hyperspy.axes.AxesManager method), 562

_remove_signal_range_in_pixels() (hyper-spy.models.model1d.Model1D method), 536

_remove_signal_range_in_pixels() (hyper-spy.models.model2d.Model2D method), 540

_remove_size_patch() (hyper-spy.drawing._widgets.line2d.Line2DWidget method), 379

_remove_widget() (hyper-spy.roi.BaseInteractiveROI method), 599

_request_user_input() (hyper-spy.samfire.Samfire method), 611

_right_changed() (hyperspy.roi.RectangularROI method), 607

_right_changed() (hyperspy.roi.SpanROI method), 608

_right_changed() (hyperspy.signal.BaseSignal method), 616

_right_value_changed() (hyper-spy.signal_tools.Signal1DCalibration method), 661

_replot() (hyperspy.signal.BaseSignal method), 616

_repr_html_() (hyperspy.axes.AxesManager method), 562

_repr_html_() (hyper-spy.misc.misc.current_component_values method), 502

_repr_html_() (hyper-spy.misc.misc.current_model_values method), 502

_repr_html_() (hyper-spy.misc.utils.DictionaryTreeBrowser method), 510

_reset() (hyperspy.signal_tools.ImageContrastEditor method), 659

_reset_line() (hyper-spy.signal_tools.SpikesRemoval method), 664

_reset_original_settings() (hyper-spy.signal_tools.ImageContrastEditor method), 659

_reset_span_selector() (hyper-spy.signal_tools.ImageContrastEditor method), 659

_reshuffle_mixed_blocks() (in module hyperspy._signals.lazy), 344

_resize() (hyper-spy.drawing._widgets.line2d.Line2DWidget method), 379

_reverse_lookup_position_widget() (hyper-spy.models.model1d.Model1D method), 536

_right_changed() (hyperspy.roi.RectangularROI method), 607

_right_changed() (hyperspy.signals.signal1d.Signal1D method), 346

_right_value_changed() (hyper-spy.signal_tools.Signal1DCalibration method), 661

_resize() (hyper-spy_drawing._widgets.line2d.Line2DWidget method), 379

_reverse_lookup_position_widget() (hyper-spy.models.model1d.Model1D method), 536

_right_changed() (hyperspy.roi.RectangularROI method), 607

_right_changed() (hyperspy.roi.SpanROI method), 608

_right_value_changed() (hyper-spy.signal_tools.Signal1DCalibration method), 661

_remove_axis() (hyperspy.signal BaseSignal method), 616

_remove_background_cli() (hyperspy.signals.signal1d.Signal1D method), 346

_remove_one_axis() (hyperspy.axes.AxesManager method), 562

_remove_axis() (hyperspy.signal.BaseSignal method), 616

_remove_background_cli() (hyperspy.signals.signal1d.Signal1D method), 346

_remove_one_axis() (hyperspy.axes.AxesManager method), 562
_set_from_widget() (hyperspy.roi.Point2DROI method), 606
_set_from_widget() (hyperspy.roi.RectangularROI method), 607
_set_from_widget() (hyperspy.roi.SpanROI method), 608
_set_imag() (hyperspy_signals.complex_signal.ComplexSignal method), 292
_set_imag() (hyperspy_signals.complex_signal.LazyComplexSignal method), 295
_set_images() (hyperspy.io_plugins.bruker.HyperHeader method), 406
_set_indices() (hyperspy.drawing.widget.DraggableWidgetBase method), 400
_set_int16() (hyperspy.io_plugins.sur.DigitalSurfHandler method), 443
_set_int32() (hyperspy.io_plugins.sur.DigitalSurfHandler method), 443
_set_metadata_and_original_metadata() (hyperspy.io_plugins.sur.DigitalSurfHandler method), 443
_set_microscope() (hyperspy.io_plugins.bruker.HyperHeader method), 406
_set_mode() (hyperspy.io_plugins.bruker.HyperHeader method), 407
_set_mpfit_parameters_info() (hyperspy.model.BaseModel method), 588
_set_name() (hyperspy.component.Component method), 567
_set_nice_description() (hyperSpy.signal.ModelManager method), 656
_set_p0() (hyperspy.model.BaseModel method), 589
_set_parameters_observer() (hyperspy.signal_tools.PeaksFinder2D method), 661
_set_patch() (hyperspy.drawing._widgets.circle.CircleWidget method), 377
_set_patch() (hyperspy.drawing._widgets.rectangles.SquareWidget method), 385
_set_patch() (hyperspy.drawing._widgets.vertical_line.VerticalLineWidget method), 386
_set_patch() (hyperspy.drawing.widget.ResizersMixin method), 402
_set_patch() (hyperspy.drawing._widgets.label.LabelWidget method), 378
_set_patch() (hyperSpy.drawing.widget.DraggableWidgetBase method), 400
_set_patch() (hyperSpy.drawing.widget.WidgetBase method), 404
_set_patch() (hyperSpy.drawing.widget.LabelWidget method), 380
_set_patch() (hyperSpy.drawing._widgets.rectangles.SquareWidget method), 385
_set_patch() (hyperSpy.drawing._widgets.vertical_line.VerticalLineWidget method), 386
_set_patch() (hyperSpy.drawing.widget.ResizersMixin method), 402
_set_patch() (hyperSpy.drawing._widgets.label.LabelWidget method), 378
_set_patch() (hyperSpy.drawing.widget.DraggableWidgetBase method), 400
_set_patch() (hyperSpy.drawing.widget.Widget1DBase method), 403
_set_phase() (hyperSpy_signals.complex_signal.ComplexSignal method), 292
_set_phase() (hyperSpy_signals.complex_signal.LazyComplexSignal method), 296
_set_position() (hyperSpy.drawing._widgets.label.LabelWidget method), 378
_set_position() (hyperSpy.drawing._widgets.scalebar.ScaleBar method), 385
_set_position() (hyperSpy.drawing.widget.DraggableWidgetBase method), 400
_set_position() (hyperSpy.drawing.widget.Widget1DBase method), 403
_set_quantity() (hyperSpy.axes.UnitConversion method), 566
_set_range() (hyperSpy.drawing._widgets.range.ModifiableSpanSelector method), 381
_set_real() (hyperSpy_signals.complex_signal.ComplexSignal method), 292
_set_real() (hyperSpy_signals.complex_signal.LazyComplexSignal method), 296
_set_resizers() (hyperSpy.drawing.widget.ResizersMixin method), 402
_set_signal_range_in_pixels() (hyperSpy.models.model1d.Model1D method), 536
_set_signal_range_in_pixels() (hyperSpy.models.model2d.Model2D method), 540
_set_size() (hyperSpy.drawing._widgets.circle.CircleWidget method), 377
_set_size() (hyperSpy.drawing._widgets.label.LabelWidget method), 378
_set_size() (hyperSpy.drawing._widgets.line2d.Line2DWidget method), 380

696 Index
_set_size() (hyper- spy.drawing._widgets.rectangles.RectangleWidget method), 383
_setup() (hyperspy.learn.rpca.ORPCA method), 464
_setup() (hyperspy.learn.ornmf.ORNMF method), 461
_setup() (hyperspy.samfire.Samfire method), 611
_setup() (hyperspy.utils.parallel_pool.ParallelPool method), 551
_setup_ipyparallel() (hyperspy.io_plugins.sur.DigitalSurfHandler method), 443
_setup_multiprocessing() (hyperspy.utils.parallel_pool.ParallelPool method), 551
_setup_vfs() (spy.io_plugins.bruker.SFS_reader method), 409
_shift1D() (in module hyperspy_signals.signal1D), 357
_show_derivative_histogram_fired() (hyperspy.signal_tools.SpikesRemoval method), 664
_show_help_fired() (hyper spy.signal_tools.ImageContrastEditor method), 659
_signal_dimension (hyper spy_signals.complex_signal1d.ComplexSignal1D attribute), 297
_signal_dimension (hyper spy_signals.complex_signal1d.LazyComplexSignal1D attribute), 297
_signal_dimension (hyper spy_signals.complex_signal2d.Complex2Dmixin attribute), 297
_signal_dimension (hyper spy_signals.signal1d.Signal1D attribute), 346
_signal_dimension (hyper spy_signals.signal2d.Signal2D attribute), 359
_signal_shape_in_array() (hyper spy_axes.AxesManager property), 562
_signal_type (hyper spy_signals.dielectric_function.DielectricFunction_mixin attribute), 301
_signal_type (hyperspy_signals.eds.EDS_mixin attribute), 303
_signal_type (hyperspy_signals.eds_sem.EDSSEM_mixin attribute), 312
_signal_type (hyper spy_signals.eds_tem.EDSTEM_mixin attribute), 315
_signal_type (hyper spy_signals.eels.EELSSpectrum_mixin attribute), 321
_signal_type (hyper
_twin_function_expr (hyper-
spy.component.Parameter attribute), 572

_update_data() (hyper-
spy.drawing.image.ImagePlot method), 387

_update_database() (hyper-
spy.samfire_utils.strategy.GlobalStrategy
method), 547

_update_database() (hyper-
spy.samfire_utils.strategy.LocalStrategy
method), 548

_update_doc() (hyperspy.events.Events method), 578

_update_free_parameters() (hyper-
spy.component.Component method), 567

_update_line() (hyper-
spy.signal_tools.BackgroundRemoval method), 657

_update_marker() (hyper-
spy.samfire_utils.strategy.GlobalStrategy
method), 547

_update_marker() (hyper-
spy.samfire_utils.strategy.LocalStrategy
method), 548

_update_max_index() (hyper-
spy.axes.AxesManager method), 562

_update_patch_geometry() (hyper-
spy.drawing._widgets.circle.CircleWidget
method), 377

_update_patch_geometry() (hyper-
spy.drawing._widgets.line2d.Line2DWidget
method), 380

_update_patch_geometry() (hyper-
spy.drawing._widgets.range.RangeWidget
method), 382

_update_patch_geometry() (hyper-
spy.drawing._widgets.rectangles.RectangleWidget
method), 383

_update_patch_position() (hyper-
spy.drawing.widget.DraggableWidgetBase
method), 400

_update_patch_position() (hyper-
spy.drawing.widget.ResizableDraggableWidgetBase
method), 401

_update_patch_position() (hyper-
spy.drawing.widget.Widget2DBase
method), 403

_update_patch_position() (hyper-
spy.drawing._widgets.circle.CircleWidget
method), 377

_update_patch_position() (hyper-
spy.drawing._widgets.horizontal_line.HorizontalLineWidget
method), 377

_update_patch_position() (hyper-
spy.drawing._widgets.label.LabelWidget
method), 378

_update_patch_position() (hyper-
align2D() (hyperspy.signals.signal2d.Signal2D method), 359
align_zero_loss_peak() (hyperspy.signals.eels.EELSSpectrum_mixin method), 321
all_warnings() (in module hyperspy.misc.test_utils), 505
alpha() (hyperspy.drawing.widget.WidgetBase property), 404
amari() (in module hyperspy.misc.machine_learning.tools), 486
amplitude() (hyperspy.signals.complex_signal.ComplexSignal_mixin property), 293
angle() (hyperspy.signals.complex_signal.ComplexSignal method), 292
angle() (hyperspy.signals.complex_signal.ComplexSignal_mixin method), 293
angle() (hyperspy.signals.complex_signal.LazyComplexSignal method), 296
angle() (hyperspy.roi.Line2DROI method), 603
angle_between() (in module hyperspy.drawing._widgets.line2d), 380
animate_legend() (in module hyperspy.drawing.utils), 392
antisymmetrize() (in module hyperspy.misc.math_tools), 500
anyfloatin() (in module hyperspy.misc.math_tools), 500
aperture_function() (in module hyperspy.misc.holography.reconstruct), 481
append() (hyperspy.model.BaseModel method), 586, 589
append() (hyperspy.models.eelsmodel.EELSMulti method), 524
append() (hyperspy.models.model1d.Model1D method), 534, 536
append() (hyperspy.models.model2d.Model2D method), 540
append() (hyperspy.samfire.Samfire method), 610, 611
append() (hyperspy.samfire.StrategyList method), 612
append2pathname() (in module hyperspy.misc.io.tools), 485
apply() (hyperspy.models.model1d.ComponentFit method), 533
apply() (hyperspy.signal_tools.BackgroundRemoval method), 657
apply() (hyperspy.signal_tools.ButterworthFilter method), 658
apply() (hyperspy.signal_tools.ImageContrastEditor method), 659
apply() (hyperspy.signal_tools.IntegrateArea method), 660
apply() (hyperspy.signal_tools.Signal1DCalibration method), 661
apply() (hyperspy.signal_tools.SmoothingLowess method), 662
apply() (hyperspy.signal_tools.SmoothingSavitzkyGolay method), 663
apply() (hyperspy.signal_tools.SmoothingTV method), 663
apply() (hyperspy.signal_tools.SpikesRemoval method), 664
apply_apodization() (hyperspy.signal.BaseSignal method), 619
apply() (hyperspy.signal_tools.SmoothingLowess method), 662
apply() (hyperspy.signal_tools.SmoothingSavitzkyGolay method), 663
apply() (hyperspy.signal_tools.SmoothingTV method), 663
apply() (hyperspy.signal_tools.SpikesRemoval method), 664
apply_apodization() (hyperspy.signal.BaseSignal method), 619
Index
calibrate_energy_axis() (hyperspy.models.edsmodel.EDSModel method), 519

calibrate_xray_lines() (hyperspy.models.edsmodel.EDSModel method), 519

centre (hyperspy.learn.mva.LearningResults attribute), 446

centre_colormap_values() (in module hyperspy.drawing.utils), 392

change_dtype() (hyperspy_signals.complex_signal.ComplexSignal_mixin method), 293

change_dtype() (hyperspy_signals.lazy.LazySignal method), 338

change_dtype() (hyperspy._signals.complex_signal.ComplexSignal_mixin method), 293

change_strategy() (hyperspy.samfire.Samfire method), 610, 611

change_timestep() (hyperspy.samfire_utils.samfire_worker.Worker method), 546

check_btn_state() (hyperspy.signal_tools.EdgesRange method), 658

check_closing_plot() (in module hyperspy.misc.test_utils), 506

check_data_tag_delimiter() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 411

check_index_valid() (hyperspy.io_plugins.brucker.BCF_reader method), 405

check_long_string() (in module hyperspy.misc.export_dictionary), 491

check_random_state() (in module hyperspy.misc.math_tools), 500

check_running_tests_in_CI() (in module hyperspy.misc.test_utils), 506

check_that_flags_make_sense() (in module hyperspy.misc.export_dictionary), 491

chisq (hyperspy.model.BaseModel attribute), 586

chisq (hyperspy.models.model1D.Model1D attribute), 533

chisq (hyperspy.models.model2D.Model2D attribute), 539

CircleROI (class in hyperspy.roi), 601

CircleWidget (class in hyperspy.drawing._widgets.circle), 376

CL_get_mass_thickness() (hyperspy_signals.Eds_TEM.EDSTEM_mixin method), 314

clean() (hyperspy.samfire_utils.strategy.GlobalStrategy method), 547

clean() (hyperspy.samfire_utils.strategy.LocalStrategy method), 446

close() (hyperspy.drawing.figure.BlittedFigure method), 386

close() (hyperspy.drawing.marker.MarkerBase method), 388

close() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 389

close() (hyperspy.drawing.signal1D.Signal1DLine method), 391

close() (hyperspy.drawing.tiles.HistogramTilePlot method), 392

close() (hyperspy.drawing.widget.WidgetBase method), 404

close() (hyperspy.signal_tools.ImageContrastEditor method), 659

close() (hyperspy.signal_tools.PeaksFinder2D method), 661

close() (hyperspy.signal_tools.Smoothing method), 662

close_file() (hyperspy_signals.lazy.LazySignal method), 339

close_navigator_plot() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 389

closest_nice_number() (in module hyperspy.misc.math_tools), 500

closest_power_of_two() (in module hyperspy.misc.math_tools), 513

cluster_algorithm (hyperspy.learn.mva.LearningResults attribute), 446

cluster_analysis() (hyperspy.learn.mva.MVA method), 451

cluster_centers (hyperspy.learn.mva.LearningResults attribute), 446

cluster_centers_estimated (hyperspy.learn.mva.LearningResults attribute), 446

cluster_labels (hyperspy.learn.mva.LearningResults attribute), 446

cluster_membership (hyperspy.learn.mva.LearningResults attribute), 446

cluster_metric (hyperspy.learn.mva.LearningResults attribute), 446

cluster_metric_data (hyperspy.learn.mva.LearningResults attribute), 446
Index 705
copy() (hyperspy.signal.BaseSignal method), 621

copy_slice_from_whitelist() (in module hyperspy.misc.slicing), 504

correct_INCA_format() (in module hyperspy.io_plugins.ripple), 437

count (hyperspy.samfire.Samfire attribute), 611

create_axes() (hyperspy.axes.AxesManager method), 563

create_axis() (hyperspy.drawing.image.ImagePlot method), 387

correct_INCA_format() (in module hyperspy.io_plugins.ripple), 437

correct_INCA_format() (in module hyperspy.io_plugins.ripple), 437

crop() (hyperspy.signal.BaseSignal method), 621

crop_decomposition_dimension() (hyperspy.learn.mva.LearningResults method), 446

crop_image() (hyperspy._signals.signal2d.Signal2D method), 360

crop_signal1D() (hyperspy._signals.signal1d.Signal1DFigure method), 390

crop_background_line() (hyperspy.signal_tools.BackgroundRemoval method), 657

crop_right_axis() (hyperspy.drawing.signal1d.Signal1DFigure method), 390

crop_left_axis() (hyperspy.drawing.signal1d.Signal1DFigure method), 390

crop_decomposition() (hyperspy._signals.eds_tem.EDSTEM_mixin method), 315

crop_decomposition() (hyperspy._signals.lazy.LazySignal method), 339

crop_decomposition_algorithm (hyperspy.learn.mva.LearningResults attribute), 447

decrease_size() (hyperspy.drawing._widgets.circle.CircleWidget method), 377

decrease_size() (hyperspy.drawing.widget.ResizableDraggableWidgetBase method), 401

deepcopy() (hyperspy.axes.AxesManager method), 563

deepcopy() (hyperspy.misc.utils.DictionaryTreeBrowser method), 511

deepcopy() (hyperspy.signal.BaseSignal method), 622

default_traits_view() (hyperspy.component.Parameter method), 572

default_value (hyperspy.component.NoneFloat attribute), 570

data (hyperspy.io_plugins.semper_unf.SemperFormat attribute), 439

data() (hyperspy.signal.BaseSignal property), 622

dataAxis (class in hyperspy.axes), 564

D
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DenseSliceCOO</td>
<td>dense_slice COO (class in <code>hypperspy.misc.io.fei_stream_readers</code>), 483</td>
<td></td>
</tr>
<tr>
<td>density_of_mixture()</td>
<td>density_of_mixture (in module <code>hypperspy.misc.material</code>), 498</td>
<td></td>
</tr>
<tr>
<td>deprecation_warning()</td>
<td>deprecation_warning (in module <code>hypperspy.misc.utils</code>), 513</td>
<td></td>
</tr>
<tr>
<td>derivative()</td>
<td>derivative (hyperspy.signal.BaseSignal method), 622</td>
<td></td>
</tr>
<tr>
<td>DerivativeTextHandler</td>
<td>derivativeTextHandler (class in <code>hypperspy.signal_tools</code>), 658</td>
<td></td>
</tr>
<tr>
<td>DerivativeTextParameters</td>
<td>derivativeTextParameters (class in <code>hypperspy.signal_tools</code>), 658</td>
<td></td>
</tr>
<tr>
<td>detect_spike()</td>
<td>detect_spike (hyperspy.signal_tools.SpikeRemoval method), 664</td>
<td></td>
</tr>
<tr>
<td>dict2hdf5group()</td>
<td>dict2hdfgroup (in module <code>hypperspy.io_plugins.hspy</code>), 427</td>
<td></td>
</tr>
<tr>
<td>dict2marker()</td>
<td>dict2marker (in module <code>hypperspy.drawing.marker</code>), 389</td>
<td></td>
</tr>
<tr>
<td>dict2sarray()</td>
<td>dict2sarray (in module <code>hypperspy.misc.array_tools</code>), 487</td>
<td></td>
</tr>
<tr>
<td>dict2signal()</td>
<td>dict2signal (in module <code>hypperspy.io</code>), 582</td>
<td></td>
</tr>
<tr>
<td>dictionaries</td>
<td>dictionaries (hypperspy.io_plugins.emd.EMD_NCEM attribute), 423</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dictionaries (hypperspy.io_plugins.emd.FeiEMDReader attribute), 424</td>
<td></td>
</tr>
<tr>
<td>dictionarize()</td>
<td>dictionarize (in module <code>hypperspy.io_plugins.bruker</code>), 410</td>
<td></td>
</tr>
<tr>
<td>dictionary_from_template()</td>
<td>dictionary_from_template (in module <code>hypperspy.defaults_parser</code>), 575</td>
<td></td>
</tr>
<tr>
<td>DictionaryTreeBrowser</td>
<td>DictionaryTreeBrowser (class in <code>hypperspy.misc.utils</code>), 509</td>
<td></td>
</tr>
<tr>
<td>DielectricFunction</td>
<td>DielectricFunction (class in <code>hypperspy.misc.electric_function</code>), 300</td>
<td></td>
</tr>
<tr>
<td>DielectricFunction_mixin</td>
<td>DielectricFunction_mixin (class in <code>hypperspy.misc.electric_function</code>), 301</td>
<td></td>
</tr>
<tr>
<td>diff()</td>
<td>diff (hyperspy.signal.lazy.LazySignal method), 340</td>
<td></td>
</tr>
<tr>
<td>diff()</td>
<td>diff (hyperspy.signal.BaseSignal method), 622</td>
<td></td>
</tr>
<tr>
<td>diff_model2plot()</td>
<td>diff_model2plot (hyperspy.signal_tools.Smoothing method), 662</td>
<td></td>
</tr>
<tr>
<td>diff_model2plot()</td>
<td>diff_model2plot (hyperspy.signal_tools.SmoothingSavitzkyGolay method), 663</td>
<td></td>
</tr>
<tr>
<td>DigitalMicrographReader</td>
<td>DigitalMicrographReader (class in <code>hypperspy.io_plugins.digital_micrograph</code>), 411</td>
<td></td>
</tr>
<tr>
<td>DigitalSurfHandler</td>
<td>DigitalSurfHandler (class in <code>hypperspy.io_plugins.sur</code>), 441</td>
<td></td>
</tr>
<tr>
<td>dimension_array_dtype()</td>
<td>dimension_array_dtype (in module <code>hypperspy.io_plugins.fei</code>), 426</td>
<td></td>
</tr>
<tr>
<td>disable_adjust_position()</td>
<td>disable_adjust_position (hyperspy.models.model2d.Model2D method), 537</td>
<td></td>
</tr>
<tr>
<td>disable_adjust_position()</td>
<td>disable_adjust_position (hyperspy.models.model1d.Model1D method), 533</td>
<td></td>
</tr>
<tr>
<td>disable_background()</td>
<td>disable_background (hyperspy.models.eelsmodel.EELSMaster method), 524</td>
<td></td>
</tr>
<tr>
<td>disable_edges()</td>
<td>disable_edges (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disable_fine_structure()</td>
<td>disable_fine_structure (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disable_free_onset_energy()</td>
<td>disable_free_onset_energy (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disable_plot_components()</td>
<td>disable_plot_components (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disable_plot_components()</td>
<td>disable_plot_components (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disable_xray_lines()</td>
<td>disable_xray_lines (hyperspy.models.eelsmodel.EELSMaster method), 525</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.drawing._widgets.RangeWidget method), 382</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.drawing.ImagePlot method), 387</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.drawing.widget.WidgetBase method), 404</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hyperspy.events.Event method), 576</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.signal_tools.BackgroundRemoval method), 657</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.signal_tools.PeaksFinder2D method), 661</td>
<td></td>
</tr>
<tr>
<td>disconnect()</td>
<td>disconnect (hypperspy.signal_tools.WidgetBase method), 404</td>
<td></td>
</tr>
<tr>
<td>DM3DataTypeError</td>
<td>DM3DataTypeError, 579</td>
<td></td>
</tr>
<tr>
<td>DM3FileVersionError</td>
<td>DM3FileVersionError, 579</td>
<td></td>
</tr>
<tr>
<td>DM3TagError</td>
<td>DM3TagError, 579</td>
<td></td>
</tr>
<tr>
<td>DM3TagIDError</td>
<td>DM3TagIDError, 579</td>
<td></td>
</tr>
<tr>
<td>DM3TagTypeError</td>
<td>DM3TagTypeError, 579</td>
<td></td>
</tr>
<tr>
<td>dof (hyperspy.model.BaseModel attribute), 586</td>
<td>dof (hyperspy.models.model1d.Model1D attribute), 533</td>
<td></td>
</tr>
<tr>
<td>dof (hyperspy.models.model2d.Model2D attribute), 539</td>
<td>dof (hyperspy.models.model2d.Model2D attribute), 539</td>
<td></td>
</tr>
<tr>
<td>Doniach (class in hyperspy_components.doniach), 240</td>
<td>Doniach (class in hyperspy_components.doniach), 240</td>
<td></td>
</tr>
<tr>
<td>DoublePowerLaw</td>
<td>DoublePowerLaw (class in hyperspy_components.eels_double_power_law), 248</td>
<td></td>
</tr>
<tr>
<td>DraggableWidgetBase</td>
<td>DraggableWidgetBase (class in hyperspy.drawing.widget), 399</td>
<td></td>
</tr>
</tbody>
</table>
draw() (hyperspy.signal_tools.LineInSignal1D method), 661
draw_patch() (hyperspy.drawing.widgets.range.ModifiableSpanSelector method), 381
draw_patch() (hyperspy.drawing.widget.WidgetBase method), 404
dtype() (hyperspy.io_plugins.digital_micrograph.ImageObject property), 412
dummy() (hyperspy.drawing.widgets.range.ModifiableSpanSelector method), 381
dummy_context_manager() (in module hyperspy.misc.utils), 513
dump_dictionary() (in module hyperspy.misc.io.tools), 485

e (hyperspy._components.eels_cl_edge.EELSCLEdge property), 247
effective_angle (hyperspy.misc.eels.effective_angle), 476
effective_angle() (in module hyperspy.misc.eels.effective_angle), 476
effective_Z() (in module hyperspy.misc.physics_tools), 502
electron_range() (in module hyperspy.misc eds.utils), 470
element_symbol() (in module hyperspy.io_plugins.phenom), 436
ElidReader (class in hyperspy.io_plugins.phenom), 435
ellipticity (hyperspy_components.gaussian2d.Gaussian2D attribute), 259
ellipticity() (hyperspy_components.gaussian2d.Gaussian2D property), 261
EMD (class in hyperspy.io_plugins.emd), 421
EMD_NCEM (class in hyperspy.io_plugins.emd), 423
emi_reader() (in module hyperspy.io_plugins.fei), 426
emixml2dtb() (in module hyperspy.io_plugins.fei), 426
enable_adjust_position() (hyperspy.models.model1d.Model1D method), 537
enable_adjust_position() (hyperspy.models.model2d.Model2D method), 540
enable_background() (hyperspy.models.edstemmodel.EELSModel method), 525
enable_edges() (hyperspy.models.edstemmodel.EELSModel method), 525
enable_fine_structure() (hyperspy.models.edstemmodel.EELSModel method), 526
enable_free_onset_energy() (hyperspy.models.edstemmodel.EELSModel method), 526
enable_plot_components() (hyperspy_components.eels_arctan, 244

Index
spy.model.BaseModel method, 590

spy.models.model1d.Model1D (hyper-
method), 537

spy.models.edsmodel.EDSM
e(n), 520

energy_axis
spy.misc.eels.hartree_slater_gos.HartreeSlaterGOS
estimate_parameters() (hyper-
attribute), 477

energy_axis
spy.misc.eels.hydrogenic_gos.HydrogenicGOS
estimate_parameters() (hyper-
attribute), 478

energy_onset
spy.misc.eels.hartree_slater_gos.HartreeSlaterGOS
estimate_parameters() (hyper-
attribute), 478

energy_onset
spy.misc.eels.hydrogenic_gos.HydrogenicGOS
estimate_parameters() (hyper-
attribute), 478

energy_to_channel() (hyper-
spy.io_plugins.bruker.EDXSpectrum method, 406

energy_to_channel() (hyper-
spy.io_plugins.bruker.edsmodel.EDSM
e method, 520

ensure_directory() (in module hyper-
spy.misc.io.tools), 485

ensure_parameters_in_bounds() (hyper-
spy.model.BaseModel method, 590

ensure_unicode() (in module hyperspy.misc.utils), 513

Erf (class in hyperspy._components.error_function), 251

estimate_background_windows() (hyper-
spy.signals.eds.EDS_mixin method, 304

estimate_elbow_position() (hyper-
spy.learn.mva.MVA method, 455

estimate_fringe_contrast_fourier() (in module hyperspy.misc.holography.tools), 482

estimate_image_shift() (in module hyperspy.signals.signal2d), 364

estimate_integration_windows() (hyper-
spy.signals.eds.EDS_mixin method, 305

estimate_map_channels() (hyper-
spy.io_plugins.bruker.HyperHeader method, 407

estimate_map_depth() (hyper-
spy.io_plugins.bruker.HyperHeader method, 407

estimate_number_of_clusters() (hyper-
spy.learn.mva.MVA method, 455

estimate_parameters() (hyper-
spy_components.doniach.Doniach method), 242

estimate_parameters() (hyper-
spy_components.exponential.Exponential method), 254

estimate_parameters() (hyper-
spy_components.gaussian.Gaussian method), 258

estimate_parameters() (hyper-
spy_components.gaussianhf.GaussianHF method), 263

estimate_parameters() (hyper-
spy_components.lorentzian.Lorentzian method), 269

estimate_parameters() (hyper-
spy_components.offset.Offset method), 270

estimate_parameters() (hyper-
spy_components.pes_voigt.PESVoigt method), 272

estimate_parameters() (hyper-
spy_components.polynomial.Polynomial method), 275

estimate_parameters() (hyper-
spy_components.polynomial_deprecated.Polynomial method), 276

estimate_parameters() (hyper-
spy_components.power_law.PowerLaw method), 278

estimate_parameters() (hyper-
spy_components.skew_normal.SkewNormal method), 284

estimate_parameters() (hyper-
spy_components.split_pvoigt.SplitVoigt method), 285

estimate_parameters() (hyper-
spy_components.voigt.Voigt method), 288

estimate_peak_width() (hyper-
spy_signals.signal1d.Signal1D method, 348

estimate_poissonian_noise_variance() (hyper-
spy_signals.BaseSignal method, 623

estimate_shift1D() (hyper-
spy_signals.signal1d.Signal1D method), 348

estimate_shift2D() (hyper-
spy_signals.signal2d.Signal2D method), 360

estimate_sideband_position() (hyper-
spy_signals.holography.HologramImage method, 331

estimate_sideband_position() (in module hyperspy.misc.holography.reconstruct), 481

estimate_sideband_size() (hyper-
spy_signals.hologram_image.HologramImage method), 709
method), 332
estimate_sideband_size() (in module hyper-
spy.misc.holography.reconstruct), 481
estimate_thickness() (hyper-
spy.signals.eels.EELSSpectrumMixin method), 324
estimate_variance_parameters() (in module hyper-
spy.misc.eels.tools), 479
estimate_zero_loss_peak_centre() (hy-
perspy.signals.eels.EELSSpectrumMixin method), 325
estimated_number_of_clusters (hyper-
spy.learn.mva.LearningResults attribute), 447
Event (class in hyperspy.events), 575
Events (class in hyperspy.events), 578
EventSuppressor (class in hyperspy.events), 577
exp_decay() (in module hyper-
spy.samfire_utils.local_strategies), 543
explained_variance (hyper-
spy.learn.mva.LearningResults attribute), 447
explained_variance_ratio (hyper-
spy.learn.mva.LearningResults attribute), 447
Exponential (class in hyper-
spy._components.exponential), 252
export() (hyperspy.component.Component method), 568
export() (hyperspy.component.Parameter method), 571, 572
export() (hyperspy.misc.utils.DictionaryTreeBrowser method), 511
export_bss_results() (hyper-
spy.signal.MVATools method), 645
export_cluster_results() (hyper-
spy.signal.MVATools method), 647
export_decomposition_results() (hyper-
spy.signal.MVATools method), 648
export_results() (hyperspy.model.BaseModel method), 587, 590
export_results() (hyperspy.models.model1d.Model1D method), 534
export_to_dictionary() (in module hyper-
spy.misc.export_dictionary), 491
Expression (class in hyper-
spy._components.expression), 254
ext_bounded (hyperspy.component.Parameter attribute), 570
ext_bounded() (hyperspy.component.Parameter property), 573
ext_force_positive() (hyper-
spy.component.Parameter property), 573
extend() (hyperspy.model.BaseModel method), 586, 591
extend() (hyperspy.models.model1d.Model1D method), 534
extend() (hyperspy.models.model2d.Model2D method), 540
extend() (hyperspy.samfire.Samfire method), 610, 611
extend() (hyperspy.samfire.StrategyList method), 612
F
factors (hyperspy.learn.mva.LearningResults attribute), 447
family_symbol() (in module hyper-
spy.io_plugins.phenom), 436
FancySlicing (class in hyperspy.misc.slicing), 503
FeiEMDReader (class in hyperspy.io_plugins.emd), 424
FeiSpectrumStream (class in hyper-
spy.io_plugins.emd), 424
fetch() (hyperspy.component.Parameter method), 573
fetch_stored_values() (hyper-
spy.component.Parameter method), 568
fetch_stored_values() (hyperspy.model.BaseModel method), 586, 591
fetch_stored_values() (hyperspy.models.model1d.Model1D method), 534
fetch_values_from_array() (hyper-
spy.component.Parameter method), 568
fetch_values_from_array() (hyper-
spy.model.BaseModel method), 591
fft() (hyperspy.signal.BaseSignal method), 624
fft_correlation() (in module hyper-
spy_signals.signal2d), 365
file_reader() (in module hyper-
spy.io_plugins.blockfile), 405
file_reader() (in module hyper-
spy.io_plugins.bruker), 410
file_reader() (in module hyper-
spy.io_plugins.dens), 411
file_reader() (in module hyper-
spy.io_plugins.digital_micrograph), 413
file_reader() (in module hyper-
spy.io_plugins.edax), 414
file_reader() (in module hyperspy.io_plugins.emd), 425
file_reader() (in module hyperspy.io_plugins.empad), 426
file_reader() (in module hyperspy.io_plugins.fei), 426
file_reader() (in module hyper-
spy.io_plugins.hspy), 427
file_reader() (in module hyperspy.io_plugins.image), 428
file_reader() (in module hyperspy.io_plugins.mrc), 428
file_reader() (in module hyperspy.io_plugins.msa), 428
file_reader() (in module hyperspy.io_plugins.netcdf), 429
file_reader() (in module hyperspy.io_plugins.nexus), 432
file_reader() (in module hyperspy.io_plugins.phenom), 436
file_reader() (in module hyperspy.io_plugins.protochips), 437
file_reader() (in module hyperspy.io_plugins.ripple), 437
file_reader() (in module hyperspy.io_plugins.semper_unf), 441
file_reader() (in module hyperspy.io_plugins.sur), 444
file_reader() (in module hyperspy.io_plugins.tif), 444
file_version() (in module hyperspy.defaults_parser), 575
file_writer() (in module hyperspy.io_plugins.blockfile), 405
file_writer() (in module hyperspy.io_plugins.emd), 425
file_writer() (in module hyperspy.io_plugins.hspy), 427
file_writer() (in module hyperspy.io_plugins.image), 428
file_writer() (in module hyperspy.io_plugins.msa), 428
file_writer() (in module hyperspy.io_plugins.nexus), 433
file_writer() (in module hyperspy.io_plugins.ripple), 438
file_writer() (in module hyperspy.io_plugins.semper_unf), 441
file_writer() (in module hyperspy.io_plugins.tif), 445
filename (hyperspy.io_plugins.bruker.SFS_reader attribute), 408
filter_butterworth() (hyperspy.signals.signal1d.Signal1D method), 349
find() (hyperspy.signal_tools.SpikesRemoval method), 664
find_dataset_paths() (hyperspy.io_plugins.emd.EMD_NCENM method), 423
find_local_max() (in module hyperspy.utils.peakfinders2D), 553
find_next_data_tag() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 411
find_next_tag() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 411
find_peaks() (hyperspy.signals.signal2d.Signal2D method), 361
find_peaks1D_ohaver() (hyperspy.signals.signal1d.Signal1D method), 349
find_peaks_dog() (in module hyperspy.utils.peakfinders2D), 553
find_peaks_log() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_max() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_minmax() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_ohaver() (in module hyperspy.signals.signal1d), 357
find_peaks_stat() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_xc() (in module hyperspy.utils.peakfinders2D), 555
find_peaks_zaefferer() (in module hyperspy.utils.peakfinders2D), 556
find_subclasses() (in module hyperspy.misc.utils), 513
fine_structure_active (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 247
fine_structure_active() (hyperspy_components.eels_cl_edge.EELSCLEdge property), 247
fine_structure_coeff (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 246
fine_structure_coeff_to_txt() (hyperspy_components.eels_cl_edge.EELSCLEdge method), 247
fine_structure_smoothing (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 246
fine_structure_width() (hyperspy_components.eels_cl_edge.EELSCLEdge property), 247
fit() (hyperspy.learn.ornmf.ORNMF method), 461
finish() (hyperspy.learn.rpca.ORPCA method), 464
finish() (hyperspy.learn.ornmf.ORNMF method), 461
find_next_data_tag() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 411
find_next_tag() (hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method), 411
find_peaks() (hyperspy.signals.signal2d.Signal2D method), 361
find_peaks1D_ohaver() (hyperspy.signals.signal1d.Signal1D method), 349
find_peaks_dog() (in module hyperspy.utils.peakfinders2D), 553
find_peaks_log() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_max() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_minmax() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_ohaver() (in module hyperspy.signals.signal1d), 357
find_peaks_stat() (in module hyperspy.utils.peakfinders2D), 554
find_peaks_xc() (in module hyperspy.utils.peakfinders2D), 555
find_peaks_zaefferer() (in module hyperspy.utils.peakfinders2D), 556
find_subclasses() (in module hyperspy.misc.utils), 513
fine_structure_active (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 247
fine_structure_active() (hyperspy_components.eels_cl_edge.EELSCLEdge property), 247
fine_structure_coeff (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 246
fine_structure_coeff_to_txt() (hyperspy_components.eels_cl_edge.EELSCLEdge method), 247
fine_structure_smoothing (hyperspy_components.eels_cl_edge.EELSCLEdge attribute), 246
fine_structure_width() (hyperspy_components.eels_cl_edge.EELSCLEdge property), 247
fit() (hyperspy.learn.ornmf.ORNMF method), 461
finish() (hyperspy.learn.rpca.ORPCA method), 464
fit() (hyperspy.learn.rpca.ORPCA method), 464
fit() (hyperspy.model.BaseModel method), 591
fit() (hyperspy.models.eelsmodel.EELSMETHOD method), 526
fit() (hyperspy.models.eelsmodel.EELSMETHOD method), 528
fit_component() (hyperspy.model.BaseModel method), 587
fit_component() (hyperspy.models.model1d.Model1D method), 534, 537
fit_background() (hyperspy.models.eelsmodel.EELSMETHOD method), 520
fit_background() (hyperspy.models.eelsmodel.EELSMETHOD method), 528
fit_cos_vignetting() (hyperspy_components.eels_vignetting.Vignetting method), 250
fit_edges() (hyperspy.models.eelsmodel.EELSMETHOD method), 528
fit_fine_structure() (hyperspy.models.eels_vignetting.Vignetting method), 250
fit_side_vignetting() (hyperspy_components.eels_vignetting.Vignetting method), 250
fit_sub_xray_lines_weight() (hyperspy.models.eelsmodel.EELSMETHOD method), 521
fix_background() (hyperspy.models.edsmodel.EDSModel method), 520
fix_cos_vignetting() (hyperspy._components.eels_vignetting.Vignetting method), 250
fix_edges() (hyperspy.models.eelsmodel.EELSMETHOD method), 520
fix_fine_structure() (hyperspy.models.eels_vignetting.Vignetting method), 520
fix_side_vignetting() (hyperspy._components.eels_vignetting.Vignetting method), 250
fix_sub_xray_lines_weight() (hyperspy.models.eelsmodel.EELSMETHOD method), 520
freq_array() (in module hyperspy.misc.holography.reconstruct), 482
from_signal() (hyperspy.io_plugins.semper_unf.SemperFormat class method), 440
fsdict() (in module hyperspy.misc.utils), 513
FUNC_A (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_B (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_MOVE (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_NONE (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_RESIZE (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_ROTATE (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
FUNC_SIZERS (hyperspy.drawing_widgets.line2d.Line2DWidget attribute), 379
fourier_log_deconvolution() (hyperspy.signals.eels.EELSSpectrum_mixin method), 325
fourier_ratio_deconvolution() (hyperspy.signals.eels.EELSSpectrum_mixin method), 326
free_background() (hyperspy.models.eelsmodel.EELSMETHOD method), 521
free_cos_vignetting() (hyperspy_components.eels_vignetting.Vignetting method), 250
function() (hyperspy_components.eels_cl_edge.EELSCLEdge method), 247
function() (hyperspy_components.eels_vignetting.Vignetting method), 250
function() (hyperspy_components.offset.Offset method), 270
function() (hyperspy_components.pes_core_line_shape.PESCoreLineShape method), 270
function() (hyperspy_components.pes_see.SEE method), 271
function() (hyperspy_components.pes_voigt.PESVoigt method)
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spy.drawing._widgets.circle.CircleWidget.get_centre()</td>
<td>hyperspy.drawing._widgets.circle.CircleWidget method</td>
<td>377</td>
</tr>
<tr>
<td>spy.drawing._widgets.line2d.Line2DWidget.get_centre()</td>
<td>hyperspy.drawing._widgets.line2d.Line2DWidget method</td>
<td>380</td>
</tr>
<tr>
<td>spy.signal.MVATools.get_cluster_distances()</td>
<td>hyperspy.signal.MVATools method</td>
<td>649</td>
</tr>
<tr>
<td>spy.signal.MVATools.get_cluster_labels()</td>
<td>hyperspy.signal.MVATools method</td>
<td>649</td>
</tr>
<tr>
<td>spy.signal.MVATools.get_cluster_signals()</td>
<td>hyperspy.signal.MVATools method</td>
<td>650</td>
</tr>
<tr>
<td>spy._signals.eels.EELSSpectrum_mixin.get_complex()</td>
<td>hyperspy._signals.eels.EELSSpectrum_mixin method</td>
<td>326</td>
</tr>
<tr>
<td>spy.drawing.signal1d.Signal1DLine.get_complex()</td>
<td>hyperspy.drawing.signal1d.Signal1DLine property</td>
<td>391</td>
</tr>
<tr>
<td>spy.signal.MVATools.get_decomposition_factors()</td>
<td>hyperspy.signal.MVATools method</td>
<td>650</td>
</tr>
<tr>
<td>spy.signal.MVATools.get_decomposition_loadings()</td>
<td>hyperspy.signal.MVATools method</td>
<td>650</td>
</tr>
<tr>
<td>hyperspy.learn.mva.MVA.get_decomposition_model()</td>
<td>hyperspy.learn.mva.MVA method</td>
<td>457</td>
</tr>
<tr>
<td>spy.default_header()</td>
<td>hyperspy.io_plugins.blockfile method</td>
<td>405</td>
</tr>
<tr>
<td>spy.io_plugins.protochips.ProtochipsCSV.get_dictionary()</td>
<td>hyperspy.io_plugins.protochips.ProtochipsCSV method</td>
<td>437</td>
</tr>
<tr>
<td>spy.signal.BaseSignal.get_dimensions_from_data()</td>
<td>hyperspy.signal.BaseSignal method</td>
<td>625</td>
</tr>
<tr>
<td>spy.io_plugins.digital_micrograph.DigitalMicrographReader.get_edges_near_energy()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method</td>
<td>480</td>
</tr>
<tr>
<td>spy.io_plugins.digital_micrograph.DigitalMicrographReader.get_file()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader method</td>
<td>409</td>
</tr>
<tr>
<td>hyperspy.learn.mva.MVA.get_explained_variance_ratio()</td>
<td>hyperspy.learn.mva.MVA method</td>
<td>301</td>
</tr>
<tr>
<td>spy.io_plugins.mrc.get_fei_dtype_list()</td>
<td>hyperspy.io_plugins.mrc method</td>
<td>428</td>
</tr>
<tr>
<td>hyperspy.signals.dielectric_function.DielectricFunction_mixin.get_electron_energy_loss_spectrum()</td>
<td>hyperspy.signals.dielectric_function.DielectricFunction_mixin method</td>
<td>301</td>
</tr>
<tr>
<td>hyperspy.learn.mva.MVA.get_histogram()</td>
<td>hyperspy.learn.mva.MVA method</td>
<td>664</td>
</tr>
<tr>
<td>hyperspy.io_plugins.mrc.get_interpolated_spectrum()</td>
<td>hyperspy.io_plugins.mrc method</td>
<td>664</td>
</tr>
<tr>
<td>hyperspy.io_plugins.protochips.ProtochipsCSV.get_interpolation_range()</td>
<td>hyperspy.io_plugins.protochips.ProtochipsCSV method</td>
<td>664</td>
</tr>
<tr>
<td>hyperspy.io_plugins.edax.get_ipython()</td>
<td>hyperspy.io_plugins.edax method</td>
<td>414</td>
</tr>
</tbody>
</table>
spy.misc.ipython_tools, 495

get_item() (hyperspy.misc.utils.DictionaryTreeBrowser method), 511

get_iter_and_properties() (hyperspy.io_plugins.bruker.SFSTreeItem method), 408

get_lengths() (in module hyperspy.io_plugins.fei), 427

get_line_length() (hyper-spy.drawing_widgets.line2d.Line2DWidget method), 380

get_linear_interpolation() (in module hyperspy.misc.math_tools), 500

get_lines_intensity() (hyper-spy.signals.eds.EDS_mixin method), 305

get_lines_intensity() (hyper-spy.models.edsmodel.EDSModel method), 521

get_low_loss_eels_line_scan_signal() (in module hyperspy.datasets.artificial_data), 368

get_low_loss_eels_signal() (in module hyperspy.datasets.artificial_data), 369

get_mapping() (hyper-spy.io_plugins.digital_micrograph.ImageObject method), 412

get_mapping() (in module hyperspy.io_plugins.bruker), 410

get_markers() (hyper-spy.misc.label_position.SpectrumLabelPosition method), 496

get_metadata() (hyper-spy.io_plugins.digital_micrograph.ImageObject method), 412

get_metadata_mapping() (in module hyperspy.io_plugins.tiff), 445

get_noise_variance() (hyper-spy.signal.BaseSignal method), 627

get_number_of_effective_electrons() (hyper-spy.signals.dielectric_function.DielectricFunction_mixin method), 301

get_object_package_info() (in module hyperspy.misc.utils), 513

get_parametrized_energy_axis() (hyper-spy.misc.eds.base_gos.GOSBase method), 474

get_parametrized_qaxis() (hyper-spy.misc.eds.tem.EDSTEM_mixin method), 474

gosfuncK() (hyperspy.misc.eds.hydrogenic_gos.HydrogenicGOS method), 479

gosfuncL() (hyperspy.misc.eds.hydrogenic_gos.HydrogenicGOS method), 479

get_poly5() (hyperspy.misc.eds.HydrogenicGOS method), 479

get_probe_area() (hyper-spy.signals.eds_tem.EDSTEM_mixin method), 317

get_qaxis_and_gos() (hyper-spy.misc.eds.GOSBase method), 474

get_qaxis_and_gos() (hyper-spy.misc.eds.hartree_slater_gos.HartreeSlaterGOS method), 477

get_qaxis_and_gos() (hyper-spy.misc.eds.hydrogenic_gos.HydrogenicGOS method), 478

get_signal_chunks() (in module hyperspy.io_plugins.hspy), 427

get_size_in_indices() (hyper-spy.drawing_widgets.circle.CircleWidget method), 377

get_size_in_indices() (hyper-spy.drawing.widget.ResizableDraggableWidgetBase method), 401

get_spd_dtype_list() (in module hyperspy.io_plugins.edax), 420

get_spc_dtype_list() (in module hyperspy.io_plugins.edax), 415

get_spectra_metadata() (hyper-spy.io_plugins.bruker.HyperHeader method), 407

get_std_dtype_list() (in module hyperspy.io_plugins.mrc), 428

get_take_off_angle() (hyper-spy.signals.eds.EDS_mixin method), 306

get_tvips_mapping() (in module hyperspy.io_plugins.tiff), 445

get_units_string() (hyper-spy.drawing.widgets.scalebar.ScaleBar method), 385

get_xml_info_from_emi() (in module hyperspy.io_plugins.xem), 427

GlobalStrategy (class in hyperspy.samfire_utils.strategy), 547

goodness_test (class in hyperspy.samfire_utils.goodness_of_fit_tests.test_general), 542

GOSBase (class in hyperspy.misc.eds.base_gos), 474

GosfuncK() (hyperspy.misc.eds.HydrogenicGOS method), 479

GosfuncL() (hyperspy.misc.eds.HydrogenicGOS method), 479

HyperSpy Documentation, Release 1.6.0

Index 715
grad_a() ( hyperspy_components.bleasdale.Bleasdale method), 240

grad_A() ( hyperspy_components.eels_double_power_law.DoublePowerLaw method), 250

grad_A() ( hyperspy_components.heaviside.HeavisideStep method), 265

grad_a() ( hyperspy_components.pes_core_line_shape.PESCoreLineShape method), 270

grad_A() ( hyperspy_components.pes_core_line_shape.PESCoreLineShape method), 270

grad_b() ( hyperspy_components.bleasdale.Bleasdale method), 240

grad_B() ( hyperspy_components.pes_core_line_shape.PESCoreLineShape method), 270

grad_c() ( hyperspy_components.bleasdale.Bleasdale method), 240

grad_coefficients() ( hyperspy_components.polynomial Deprecated Polynomial fit method), 276

grad_FWHM() ( hyperspy_components.pes_core_line_shape.PESCoreLineShape hyperspy.roi, 609

grad_fwhm() ( hyperspy_components.volume_plasmon_drude.VolumePlasmonDrude method), 291

grad_intensity() ( hyperspy_signals eds_sem.EDSSEMParametersUI method), 248

grad_intensity() ( hyperspy_signals eds_cl_edge.EELSCLEdge method), 248

grad_n() ( hyperspy_components.heaviside.HeavisideStep method), 265

grad_offset() ( hyperspy_components.offset.Offset method), 270

grad_one_coefficient() ( hyperspy_components.polynomial Deprecated Polynomial fit method), 276

grad_origin() ( hyperspy_components.eels_double_power_law.DoublePowerLaw method), 250

grad_origin() ( hyperspy_components.pes_core_line_shape.PESCoreLineShape method), 270

grad_origin() ( hyperspy_components.power_law.PowerLaw method), 278

grad_Phi() ( hyperspy_components.pes_see.SEE method), 271

grad_plasmon_energy() ( hyperspy_components.volume_plasmon_drude.VolumePlasmonDrude method), 291

grad_offset() ( hyperspy_components.eels_double_power_law.DoublePowerLaw method), 250

grad_shift() ( hyperspy_components.eels_double_power_law.DoublePowerLaw method), 250

grad_ratio() ( hyperspy_components.eels_double_power_law.DoublePowerLaw method), 250

grad_r() ( hyperspy_components.power_law.PowerLaw method), 279

guess_gos_path() ( in module hyperspy.defaults_parser), 575

guess_mode() ( in module hyperspy.io_plugins.bruker), 410

guess_record_by() ( in module hyperspy.io_plugins.fei), 427

guess_vertical_or_horizontal() ( in module hyperspy.roi), 609

gui() ( hyperspy_components.eels_cl_edge.EELSCLEdge method), 248

gui() ( hyperspy_signals eds_sem.EDSSEMParametersUI method), 281

gui() ( hyperspy_signals eds_cl_edge.EELSCLEdge method), 281

gui() ( hyperspy_signals eds_tm.EDSTEMParametersUI method), 281

gui() ( hyperspy_signals eds_eelstem.EELSTEMParametersUI method), 331

gui() ( hyperspy_axes.AxesManager method), 563

gui() ( hyperspy_axes.DataAxis method), 565

gui() ( hyperspy_component.Component method), 568

gui() ( hyperspy_component.Parameter method), 573

gui() ( hyperspy_defaults_parser.Preferences method), 575

gui() ( hyperspy_models.model1D.ComponentFit method), 593

gui() ( hyperspy_models.models1D.ComponentFit method), 533

gui() ( hyperspy_roi.CircleROI method), 601

gui() ( hyperspy_roi.Line2DROI method), 603

gui() ( hyperspy_roi.Point1DROI method), 605

gui() ( hyperspy_roi.Point2DROI method), 606

gui() ( hyperspy_roi.RectangularROI method), 607

gui() ( hyperspy_roi.SpanROI method), 608

gui() ( hyperspy_signal_tools.BackgroundRemoval method), 657

hyperspy.signal_tools.ButterworthFilter
HyperSpy Documentation, Release 1.6.0

method, 658
gui() (hyperspy.signal_tools.EdgesRange method), 658
gui() (hyperspy.signal_tools.ImageContrastEditor method), 659
gui() (hyperspy.signal_tools.IntegrateArea method), 660
gui() (hyperspy.signal_tools.PeaksFinder2D method), 661
gui() (hyperspy.signal_tools.SimpleMessage method), 662
gui() (hyperspy.signal_tools.SmoothingLowess method), 662
gui() (hyperspy.signal_tools.SmoothingSavitzkyGolay method), 663
gui() (hyperspy.signal_tools.SmoothingTV method), 663
gui() (hyperspy.signal_tools.SpikesRemoval method), 664
gui_adjust_contrast() (hyperSpy.drawing.image.ImagePlot method), 387
gui_navigation_sliders() (hyperSpy.axes.AxesManager method), 563
GUIs (class in hyperspy.defaults_parser), 575
gwidth() (hyperspy.components.gaussian.Gaussian property), 258

H

hann_window_nth_order() (in module hyperspy.misc.math_tools), 500
hanning2d() (in module hyperspy.signals.signal2d), 365
hanning_taper() (hyperSpy_signals.signal1d.Signal1D method), 350
HartreeSlaterGOS (class in hyperspy.misc.eels.hartree_slater_gos), 477
has_item() (hyperspy.misc.utils.DictionaryTreeBrowser method), 511
has_pool (hyperspy.samfire_utils.samfire_pool.SignalPool attribute), 544
has_pool (hyperspy.utils.parallel_pool.ParallelPool attribute), 551
has_pool() (hyperspy.utils.parallel_pool.ParallelPool property), 551
hdfgroup2dict() (in module hyperspy.io_plugins.hspy), 428
hdfgroup2signaldict() (in module hyperspy.io_plugins.hspy), 428
HEADER_DTYPE (hyperSpy.io_plugins.semper_unf.SemperFormat attribute), 440
HeavisideStep (class in hyperspy_components.heaviside), 263
height() (hyperspy_components.gaussian.Gaussian property), 258
height() (hyperspy_components.lorentzian.Lorentzian property), 269
height() (hyperspy_components.split_pvoigt.SplitVoigt property), 286
height() (hyperspy_drawing_widgets.rectangles.RectangleWidget property), 384
height() (hyperspy.roi.RectangularROI property), 607
histogram() (in module hyperspy.misc.hist_tools), 493
histogram_dask() (in module hyperspy.misc.hist_tools), 494
HistogramSegmenter (class in hyperspy.samfire_utils.segmenters.histogram), 542
HistogramStrategy (class in hyperspy.samfire_utils.global_strategies), 543
HistogramTilePlot (class in hyperspy.drawing.tiles), 392
HologramImage (class in hyperspy_signals.hologram_image), 331
homogenize_ndim() (in module hyperspy.misc.array_tools), 487
HorizontalLine (class in hyperspy.drawing_markers-horizontal_line), 370
HorizontalLineSegment (class in hyperspy.drawing_markers-horizontal_line_segment), 371
HorizontalLineWidget (class in hyperspy.drawing_widgets-horizontal_line), 377
HydrogenicGOS (class in hyperspy.misc.eels.hydrogenic_gos), 478
HyperHeader (class in hyperspy.io_plugins.bruker), 406
hyperspy module, 666
hyperspy_components module, 291
hyperspy_components.arctan module, 237
hyperspy_components.bleasdale module, 239
hyperspy_components.doniach module, 240
hyperspy_components.eels_arctan module, 243
hyperspy_components.eels_cl_edge module, 246
hyperspy_components.eels_double_power_law module, 248
<table>
<thead>
<tr>
<th>Package</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hyperspy.components.eels_vignetting</td>
<td>module, 250</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.error_function</td>
<td>module, 251</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.exponential</td>
<td>module, 252</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.expression</td>
<td>module, 254</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.gaussian</td>
<td>module, 256</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.gaussian2d</td>
<td>module, 258</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.gaussianhf</td>
<td>module, 261</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.heaviside</td>
<td>module, 263</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.logistic</td>
<td>module, 265</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.lorentzian</td>
<td>module, 267</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.offset</td>
<td>module, 270</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.pes_core_line_shape</td>
<td>module, 270</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.pes_see</td>
<td>module, 271</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.pes_voigt</td>
<td>module, 271</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.polynomial</td>
<td>module, 274</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.polynomial_deprecate</td>
<td>module, 276</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.power_law</td>
<td>module, 277</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.rc</td>
<td>module, 279</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.scalable_fixed_pattern</td>
<td>module, 280</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.skew_normal</td>
<td>module, 282</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.split_pvoigt</td>
<td>module, 285</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.voigt</td>
<td>module, 286</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.components.volume_plasmon_drude</td>
<td>module, 289</td>
<td>292</td>
</tr>
<tr>
<td>hyperspy.lazy_signals</td>
<td>module, 558</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals</td>
<td>module, 366</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.common_signal1d</td>
<td>module, 291</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.common_signal2d</td>
<td>module, 292</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.complex_signal1d</td>
<td>module, 296</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.complex_signal2d</td>
<td>module, 297</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.dielectric_function</td>
<td>module, 300</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.eds</td>
<td>module, 302</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.eds_sem</td>
<td>module, 312</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.eds_tem</td>
<td>module, 314</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.eels</td>
<td>module, 321</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.hologram_image</td>
<td>module, 331</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.lazy</td>
<td>module, 336</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.lazy_signal1d</td>
<td>module, 345</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.lazy_signal2d</td>
<td>module, 358</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.lazy</td>
<td>module, 361</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.signals.lazysignals</td>
<td>module, 366</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets</td>
<td>module, 369</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets.artificial_data</td>
<td>module, 366</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets.example_signals</td>
<td>module, 369</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.decorators</td>
<td>module, 574</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets</td>
<td>module, 574</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets</td>
<td>module, 574</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets</td>
<td>module, 574</td>
<td>370</td>
</tr>
<tr>
<td>hyperspy.datasets</td>
<td>module, 574</td>
<td>370</td>
</tr>
</tbody>
</table>
Index 719
hyperSpy Documentation, Release 1.6.0

 hyperspy.misc.tv_denoise  
   module, 506
 hyperspy.misc.utils  
   module, 509
 hyperspy.model  
   module, 586
 hyperspy.models  
   module, 541
 hyperspy.models.edsmode  
   module, 517
 hyperspy.models.edssemmodel  
   module, 523
 hyperspy.models.edstemmodel  
   module, 523
 hyperspy.models.eelsmodel  
   module, 524
 hyperspy.models.model1d  
   module, 533
 hyperspy.models.model2d  
   module, 539
 hyperspy.Release  
   module, 558
 hyperspy.roi  
   module, 598
 hyperspy.samfire  
   module, 609
 hyperspy.samfire_utils  
   module, 549
 hyperspy.samfire_utils.fit_tests  
   module, 543
 hyperspy.samfire_utils.global_strategies  
   module, 543
 hyperspy.samfire_utils.eds  
   module, 549
 hyperspy.samfire_utils.markers  
   module, 550
 hyperspy.samfire_utils.material  
   module, 550
 hyperspy.samfire_utils.model  
   module, 550
 hyperspy.samfire_utils.model_selection  
   module, 550
 hyperspy.samfire_utils.peakfinders2D  
   module, 552
 hyperspy.samfire_utils.plot  
   module, 556
 hyperspy.samfire_utils.roi  
   module, 556
 hyperspy.samfire_utils.samfire  
   module, 557
 hyperspy.samfire_utils.segmenters  
   module, 542
 hyperspy.samfire_utils.segmenters.histogram  
   module, 542
 hyperspy.samfire_utils.strategy  
   module, 547
 hyperspy.samfire_utils.weights  
   module, 543
 hyperspy.samfire_utils.weights.red_chisq  
   module, 542
 hyperspy.signal  
   module, 613
 hyperspy.signal_tools  
   module, 657
 hyperspy.signals  
   module, 665
 hyperspy.ui_registry  
   module, 666
 hyperspy.utils  
   module, 557
 hyperspy.utils.eds  
   module, 549
 hyperspy.utils.markers  
   module, 550
 hyperspy.utils.material  
   module, 550
 hyperspy.utils.model  
   module, 550
 hyperspy.utils.parallel_pool  
   module, 550
 hyperspy.utils.peakfinders2D  
   module, 552
 hyperspy.utils.plot  
   module, 556
 hyperspy.utils.roi  
   module, 556
 hyperspy.utils.samfire  
   module, 557
 hyperspy.utils.peakfinders2D  
   module, 552
 hyperspy.utils.plot  
   module, 556
 hyperspy.utils.roi  
   module, 556
 hyperspy.utils.samfire  
   module, 557
 hyperspy.utils.segmenters段落  
   module, 542
 hyperspy.utils.segmenters.histogram  
   module, 542
 hyperspy.utils.strategy  
   module, 547
 hyperspy.utils.weights  
   module, 543
 hyperspy.utils.weights.red_chisq  
   module, 542
 ifft()  
   (in module hyperspy.signal.BaseSignal method), 627
 ICLASS_DICT  
   (hyperSpy.io_plugins.semper_unf.SemperFormat attribute), 440
 ICLASS_DICT_INV  
   (hyperSpy.io_plugins.semper_unf.SemperFormat attribute), 440
 ignore_warning()  
   (in module hyperspy.misc.test_utils), 506
 imag()  
   (hyperSpy.signals.complex_signal.ComplexSignal_mixin property), 293

Index
spy.utils.parallel_pool.ParallelPool property, 552
is_on() (hyperspy.drawing.widget.WidgetBase method), 404
is_rgb() (hyperspy.signal.BaseSignal property), 630
is_rgb() (in module hyperspy.misc.rgb_tools), 503
is_rgb() (hyperspy.signal.BaseSignal property), 630
is_rgb() (in module hyperspy.misc.rgb_tools), 503
is_rgbx() (in module hyperspy.misc.rgb_tools), 503
is_span_selector_valid() (hyperspy.signal_tools.SpanSelectorInSignal1D property), 664
is_valid() (hyperspy.roi.BaseROI method), 601
is_valid() (hyperspy.roi.CircleROI method), 602
is_valid() (hyperspy.roi.Line2DROI method), 603
is_valid() (hyperspy.roi.Point1DROI method), 605
is_valid() (hyperspy.roi.Point2DROI method), 606
is_valid() (hyperspy.roi.RectangularROI method), 607
is_valid() (hyperspy.roi.SegROI method), 608
IsBZip2() (in module hyperspy.io_plugins.phenom), 436
isfloat() (in module hyperspy.misc.math_tools), 501
IsGZip() (in module hyperspy.io_plugins.phenom), 436
isiterable() (hyperspy.drawing.marker.MarkerBase method), 388
isiterable() (in module hyperspy.misc.utils), 514
ISO_format_to_serial_date() (in module hyperspy.misc.date_time_tools), 489
iterable_not_string() (in module hyperspy.misc.utils), 514
K
key2switch_right_pointer() (hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer method), 390
key_navigator() (hyperspy.axesAxesManager method), 563
key_press_handler_custom() (in module hyperspy.drawing.utils), 393
keys() (hyperspy.misc.utils.DictionaryTreeBrowser method), 512
kramers_kronig_analysis() (hyperspy.signals.eels.EELSSpectrum_mixin method), 326
L
LABEL_DTYPES (hyperspy.io_plugins.semper_unf.SemperFormat attribute), 440
LabelWidget (class in hyperspy.drawing._widgets.label), 378
lazify() (in module hyperspy.decorators), 574
lazifyTestClass() (in module hyperspy.decorators), 574
LazyComplexSignal (class in hyperspy._signals.complex_signal), 295
LazyComplexSignal1D (class in hyperspy._signals.complex_signal1d), 297
LazyComplexSignal2D (class in hyperspy._signals.complex_signal2d), 300
LazyDielectricFunction (class in hyperspy._signals.dielectric_function), 301
LazyEDSSEM Spectrum (class in hyperspy._signals.edsem), 313
LazyEDSSpectrum (class in hyperspy._signals.eds), 311
LazyEDSTEM Spectrum (class in hyperspy._signals.eds_temp), 320
LazyEELSSpectrum (class in hyperspy._signals.eels), 331
LazyHologramImage (class in hyperspy._signals.hologram_image), 335
LazySignal (class in hyperspy._signals.lazy), 336
LazySignal1D (class in hyperspy._signals.signal1d), 345
LazySignal2D (class in hyperspy._signals.signal2d), 358
LearningResults (class in hyperspy.learn.mva), 446
legend_artist() (hyperspy.signal_tools.DerivativeTextHandler method), 658
length() (hyperspy.roi.Line2DROI method), 603
Line2DROI (class in hyperspy.roi), 602
Line2DWidget (class in hyperspy.drawing._widgets.line2d), 378
line_color_rgb() (hyperspy.signal_tools.Smoothing property), 662
line_properties (hyperspy.drawing.signal1D.Signal1DLine attribute), 391
line_properties() (hyperspy.drawing.signal1D.Signal1DLine property), 391
LineInSignal1D (class in hyperspy.signal_tools), 660
LineSegment (class in hyperspy.drawing._markers.line_segment), 372
list_datasets_in_file() (in module hyperspy.io_plugins.nexus), 433
listen() (hyperspy.samfire_utils.samfire_worker.Worker method), 546
Load (class in hyperspy.signal_tools), 661
load() (hyperspy.learn.mva.LearningResults method),
Model2D (class in hyperspy.models.model2d), 539
model2plot() (hyper-
spy.signal_tools.ButterworthFilter method), 658
model2plot() (hyper-
spy.signal_tools.SmoothingLowess method), 663
model2plot() (hyper-
spy.signal_tools.SmoothingSavitzkyGolay method), 663
model2plot() (hyperspy.signal_tools.SmoothingTV method), 663
ModelComponents (class in hyperspy.model), 598
ModelManager (class in hyperspy.signal), 656
ModelManager.ModelStub (class in hyper-
spy.signal), 656
ModelSpecialSlicers (class in hyperspy.model), 598
ModifiableSpanSelector (class in hyper-
spy.drawing._widgets.range), 381
module
hyperspy, 666
hyperspy._components, 291
hyperspy._components.arctan, 237
hyperspy._components.bleasdale, 239
hyperspy._components.doniach, 240
hyperspy._components.eels_arctan, 243
hyperspy._components.eels_cl_edge, 246
hyperspy._components.eels_double_power_law, 248
hyperspy._components.eels_vignetting, 250
hyperspy._components.error_function, 251
hyperspy._components.exponential, 252
hyperspy._components.expression, 254
hyperspy._components.gaussian, 256
hyperspy._components.gaussian2d, 258
hyperspy._components.gaussianhf, 261
hyperspy._components.heaviside, 263
hyperspy._components.logistic, 265
hyperspy._components.lorentzian, 267
hyperspy._components.offset, 270
hyperspy._components.pes_core_line_shape, 270
hyperspy._components.pes_see, 271
hyperspy._components.pes_voigt, 271
hyperspy._components.pes_voigt, 271
hyperspy._components.polynomial, 274
hyperspy._components.polynomial_deprecated, 276
hyperspy._components.power_law, 277
hyperspy.drawing._markers.horizontal_line
hyperspy.drawing._markers.horizontal_line_segment
hyperspy.drawing._markers.line_segment
hyperspy.drawing._markers.point
hyperspy.drawing._markers.rectangle
hyperspy.drawing._markers.text
hyperspy.drawing._markers.vertical_line
hyperspy.drawing._markers.vertical_line_segment
hyperspy.drawing._widgets
hyperspy.drawing._widgets.circle
hyperspy.drawing._widgets.horizontal_line
hyperspy.drawing._widgets.label
hyperspy.drawing._widgets.line2d
hyperspy.drawing._widgets.range
hyperspy.drawing._widgets.rectangles
hyperspy.drawing._widgets.scalebar
hyperspy.drawing._widgets.vertical_line
hyperspy.drawing.figure
hyperspy.drawing.image
hyperspy.drawing.marker
hyperspy.drawing.mpl_he
hyperspy.drawing.mpl_hie
hyperspy.drawing.mpl_hse
hyperspy.drawing.signal
hyperspy.drawing.signalld
hyperspy.drawing.tiles
hyperspy.drawing.utils
hyperspy.drawing.widget
hyperspy.drawing.widgets
hyperspy.events
hyperspy.exceptions
hyperspy.extensions
hyperspy.interactive
hyperspy.io
hyperspy.io_plugins
hyperspy.io_plugins.empad
hyperspy.io_plugins.fei
hyperspy.io_plugins.hapy
hyperspy.io_plugins.image
hyperspy.io_plugins.mrc
hyperspy.io_plugins.msa
hyperspy.io_plugins.netcdf
hyperspy.io_plugins.nexus
hyperspy.io_plugins.phenom
hyperspy.io_plugins.protochips
hyperspy.io_plugins.ripple
hyperspy.io_plugins.sempere
hyperspy.io_plugins.sur
hyperspy.io_plugins.tif
hyperspy.io_plugins.tiff
hyperspy.learn
hyperspy.learn.mipca
hyperspy.learn.mva
hyperspy.learn.ornmf
hyperspy.learn.orthomax
hyperspy.learn rpca
hyperspy.learn.whitening
hyperspy.logger
hyperspy.misc
hyperspy.misc.array_tools
hyperspy.misc.config_dir
hyperspy.misc.date_time_tools
hyperspy.misc.eds
hyperspy.misc.eds.ffast_mac
hyperspy.misc.eds.utils
hyperspy.misc.eels
hyperspy.misc.eels.base_gos
hyperspy.misc.eels.eelsdb
hyperspy.misc.eels.effective_angle
hyperspy.misc.eels.electron_inelastic_mean_free_path
hyperspy.misc.eels.hartree_slater_gos
hyperspy.misc.eels.hydrogenic_gos
hyperspy.misc.eels.tools
hyperspy.misc.eels.utils
hyperspy.misc.holography
hyperspy.misc.holography.reconstruct
hyperspy.misc.holography.tools
hyperspy.misc.io
hyperspy.misc.io.feistream_readers
Offset (class in hyperspy._components.offset), 270

ORNMF (class in hyperspy.learn.ornmf), 461
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Caption</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ornmf()</td>
<td>hyperspy.learn.ornmf</td>
<td>462</td>
</tr>
<tr>
<td>ORPCA</td>
<td>hyperspy.learn.rpca</td>
<td>463</td>
</tr>
<tr>
<td>orphmax()</td>
<td>hyperspy.learn.orthomax</td>
<td>465</td>
</tr>
<tr>
<td>outer_nd()</td>
<td>hyperspy.misc.math_tools</td>
<td>501</td>
</tr>
<tr>
<td>output_dimension</td>
<td>LearningResults</td>
<td>447</td>
</tr>
<tr>
<td>overwrite()</td>
<td>hyperspy.misc.io_tools</td>
<td>485</td>
</tr>
<tr>
<td>overwrite_dataset()</td>
<td>hyperspy.io_plugins.hspy</td>
<td>428</td>
</tr>
<tr>
<td>pack_to_intbytes()</td>
<td>hyperspy.io_plugins.semper_unf</td>
<td>441</td>
</tr>
<tr>
<td>ParallelPool</td>
<td>hyperspy.utils.parallel_pool</td>
<td>550</td>
</tr>
<tr>
<td>Parameter</td>
<td>hyperspy.component</td>
<td>570</td>
</tr>
<tr>
<td>parametrize_GOS()</td>
<td>hyperspy.misc.eels.hydrogenic_gos</td>
<td>478</td>
</tr>
<tr>
<td>parse()</td>
<td>hyperspy.samfire_utils.samfire_pool.SamfirePool</td>
<td>544, 545</td>
</tr>
<tr>
<td>parse()</td>
<td>hyperspy.samfire_utils.samfire_worker.Worker</td>
<td>546</td>
</tr>
<tr>
<td>parse_array_definition()</td>
<td>hyperspy.io_plugins.digital_micrograph</td>
<td>411</td>
</tr>
<tr>
<td>parse_flag_string()</td>
<td>hyperspy.io_plugins.export_dictionary</td>
<td>492</td>
</tr>
<tr>
<td>parse_header()</td>
<td>hyperspy.io_plugins.digital_micrograph</td>
<td>411</td>
</tr>
<tr>
<td>parse_hypermap()</td>
<td>hyperspy.io_plugins.bruker.BCF_reader</td>
<td>405</td>
</tr>
<tr>
<td>parse_line()</td>
<td>hyperspy.io_plugins.bruker</td>
<td>410</td>
</tr>
<tr>
<td>parse_msa_string()</td>
<td>hyperspy.io_plugins.msa</td>
<td>428</td>
</tr>
<tr>
<td>parse_quantity()</td>
<td>hyperspy.misc.util</td>
<td>515</td>
</tr>
<tr>
<td>parse_ripple()</td>
<td>hyperspy.io_plugins.ripple</td>
<td>438</td>
</tr>
<tr>
<td>parse_string_definition()</td>
<td>hyperspy.io_plugins.digital_micrograph</td>
<td>411</td>
</tr>
<tr>
<td>parse_struct_definition()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader</td>
<td>411</td>
</tr>
<tr>
<td>parse_tag_group()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader</td>
<td>411</td>
</tr>
<tr>
<td>parse_tag_header()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader</td>
<td>411</td>
</tr>
<tr>
<td>parse_tags()</td>
<td>hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader</td>
<td>411</td>
</tr>
<tr>
<td>parse_TrueImageHeaderInfo()</td>
<td>hyperspy.io_plugins.fei</td>
<td>427</td>
</tr>
<tr>
<td>PeaksFinder2D</td>
<td>hyperspy.signal_tools</td>
<td>661</td>
</tr>
<tr>
<td>PESCoreLineShape</td>
<td>hyperspy._components.pes_core_line_shape</td>
<td>270</td>
</tr>
<tr>
<td>PESVoigt</td>
<td>hyperspy._components.pes_voigt</td>
<td>271</td>
</tr>
<tr>
<td>phase()</td>
<td>hyperspy._signals.complex_signal.ComplexSignal.mixin</td>
<td>293</td>
</tr>
<tr>
<td>Phi</td>
<td>hyperspy._components.pes_voigt.SEE</td>
<td>271</td>
</tr>
<tr>
<td>picker_kwargs()</td>
<td>hyperspy.drav utils</td>
<td>394</td>
</tr>
<tr>
<td>ping()</td>
<td>hyperspy.samfire_utils.samfire_worker.Worker</td>
<td>546</td>
</tr>
<tr>
<td>ping_workers()</td>
<td>hyperspy.samfire_utils.samfire_pool.SamfirePool</td>
<td>543, 545</td>
</tr>
<tr>
<td>pixels_done()</td>
<td>hyperspy.samfire.Samfire</td>
<td>611</td>
</tr>
<tr>
<td>pixels_left()</td>
<td>hyperspy.samfire.Samfire</td>
<td>611</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.complex_signal.ComplexSignal.mixin</td>
<td>293</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.complex_signal2d.Complex2D mixin</td>
<td>298</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.eels.EDS.mixin</td>
<td>307</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.eels.EELSSpectrum.mixin</td>
<td>328</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.signal1d.Signal1D</td>
<td>351</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy._signals.signal2d.Signal2D</td>
<td>362</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy.component.Component</td>
<td>568</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy.component.Parameter</td>
<td>571</td>
</tr>
<tr>
<td>plot()</td>
<td>hyperspy.drawing.image.ImagePlot</td>
<td>729</td>
</tr>
</tbody>
</table>
plot() (hyperspy.drawing.marker.MarkerBase method), 388
plot() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 389
plot() (hyperspy.drawing.signal1d.Signal1DFigure method), 390
plot() (hyperspy.drawing.signal1d.Signal1DLine method), 391
plot() (hyperspy.drawing.tiles.HistogramTilePlot method), 392
plot() (hyperspy.model.BaseModel method), 587
plot() (hyperspy.models.model1d.Model1D method), 534, 538
plot() (hyperspy.models.model2d.Model2D method), 540
plot() (hyperspy.samfire.Samfire method), 610, 612
plot() (hyperspy.models.model1d.Model1D method), 534
plot_bss_factors() (hyperspy.signal.MVATools method), 650
plot_bss_loadings() (hyperspy.signal.MVATools method), 650
plot_bss_results() (hyperspy.signal.MVATools method), 651
plot_cluster_distances() (hyperspy.signal.MVATools method), 652
plot_cluster_labels() (hyperspy.signal.MVATools method), 652
plot_cluster_metric() (hyperspy.learn.mva.MVA method), 458
plot_cluster_results() (hyperspy.signal.MVATools method), 653
plot_cluster_signals() (hyperspy.signal.MVATools method), 653
plot_cumulative_explained_variance_ratio() (hyperspy.learn.mva.MVA method), 458
plot_decomposition_factors() (hyperspy.signal.MVATools method), 654
plot_decomposition_loadings() (hyperspy.signal.MVATools method), 654
plot_decomposition_results() (hyperspy.signal.MVATools method), 655
plot_edges_label() (hyper-spy_signals.eels.EELSSpectrum_mixin method), 328
plot_every (hyperspy.samfire.Samfire attribute), 610, 612
plot_explained_variance_ratio() (hyperspy.learn.mva.MVA method), 458
plot_histogram() (hyperspy.signal_tools.ImageContrastEditor method), 660
plot_histograms() (in module hyperspy.drawing.utils), 394
plot_images() (in module hyperspy.drawing.utils), 394
plot_navigator() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 389
plot_results() (hyperspy.model.BaseModel method), 587, 594
plot_results() (hyperspy.models.model1d.Model1D method), 534
plot_RGB_map() (in module hyperspy.drawing.utils), 394
plot_scale() (hyperspy.drawing._widgets.scalebar.ScaleBar method), 385
plot_signal() (hyperspy.drawing.mpl_he.MPL_HyperExplorer method), 389
plot_signal() (hyperspy.drawing.mpl_hie.MPL_HyperImage_Explorer method), 390
plot_signal() (hyperspy.drawing.mpl_hse.MPL_HyperSignal1D_Explorer method), 390
plot_signals() (in module hyperspy.drawing.utils), 397
plot_spectra() (in module hyperspy.drawing.utils), 398
PlotConfig (class in hyperspy.defaults_parser), 575
Point (class in hyperspy.drawing._markers.point), 373
Point1DROI (class in hyperspy.roi), 604
Point2DROI (class in hyperspy.roi), 605
poissonian_noise_normalized (hyper-spy.learn.mva.LearningResults attribute), 447
Polynomial (class in hyperspy_components.polynomial), 274
Polynomial (class in hyperspy_components.polynomial_deprecated), 276
pool (hyperspy.samfire.Samfire attribute), 609, 612
pool (hyperspy.samfire_utils.samfire_pool.SamfirePool attribute), 544
pool (hyperspy.utils.parallel_pool.ParallelPool attribute), 551
pop() (hyperspy.signal.ModelManager method), 656
position (hyperspy.signal_tools.LineInSignal1D attribute), 660
position() (hyperspy.drawing.widget.DraggableWidgetBase
HyperSpy Documentation, Release 1.6.0

property, 400
post_process_RGB() (hyper-
spy.io_plugins.sur.DigitalSurfHandler method), 444
power_law_extrapolation() (hyper-
spy_signals.eels.EELSSpectrum_mixin method), 328
power_law_perc_area() (in module hyper-
spy.misc.eels.tools), 480
PowerLaw (class in hyperspy_components.power_law), 277
Preferences (class in hyperspy_defaults_parser), 575
prepare_interpolator() (hyper-
spy_components.scale_fixed_pattern.ScalableFixedPattern method), 281
prepare_workers() (hyper-
spy.samfire_utils.samfire_pool.SamfirePool method), 543, 545
print_current_values() (hyper-
spy_component.Component method), 569
print_current_values() (hyper-
spy_model.BaseModel method), 587, 595
print_current_values() (hyper-
spy.models.model1d.Model1D method), 534
print_edges_near_energy() (hyper-
spy_signals.eels.EELSSpectrum_mixin static method), 329
print_html() (in module hyperspy_misc-utils), 515
print_known_signal_types() (in module hyper-
spy_utils), 557
print_summary_statistics() (hyper-
spy.signal.BaseSignal method), 634
profile_line() (hyperspy.roi.Line2DROI static method), 603
project() (hyperspy_learn_ornmf.ORNMF method), 462
project() (hyperspy_learn_rpca.ORPCA method), 464
Properties (hyperspy_components.gaussian2d.Gaussian2D attribute), 259
ProtochipsCSV (class in hyper-
spy_io_plugins.protochips), 436
py_parse_hypermap() (in module hyper-
spy_io_plugins.bruker), 410
quantification_cliff_lorimer() (in module hyperspy_misc eds-utils), 471
quantification_cross_section() (in module hyperspy_misc eds-utils), 472
quantification_zeta_factor() (in module hyper-
spy_misc eds-utils), 472
quantify() (hyperspy_models.eelsspectrum.eELSSpectrumModel method), 529
R
radii() (hyperspy_samfire_utils.strategy.LocalStrategy property), 548
random_state (hyperspy_samfire.Samfire attribute), 610
range() (hyperspy_drawing_widgets.range.ModifiableSpanSelector property), 381
RangeWidget (class in hyper-
spy_drawing_widgets.range), 382
ratio() (in module hyperspy_misc eds-tools), 480
RC (class in hyperspy_components.re), 279
read_array() (hyper-
spy_io_plugins.digital_micrograph.DigitalMicrographReader method), 412
read_boolean() (in module hyperspy_misc io-utils_readfile), 485
read_byte() (in module hyperspy_misc io-utils_readfile), 485
read_char() (in module hyperspy_misc io-utils_readfile), 485
read_double() (in module hyperspy_misc io-utils_readfile), 485
read_elements() (hyper-
spy_misc eds_base_gos.GOSBase method), 474
read_emd_version() (in module hyper-
spy_io_plugins.emd), 426
read_file() (hyperspy_io_plugins.emd.EMD_NCEM method), 423
read_file() (hyper-
spy_io_plugins.emd.FeiEMDReader method), 424
read_float() (in module hyperspy_misc io-utils_readfile), 485
read_l_or_q() (hyper-
spy_io_plugins.digital_micrograph.DigitalMicrographReader property), 412
Q
qaxis (hyperspy_misc eds_hartree_slater_gos.HartreeSlaterGOS attribute), 478
qaxis (hyperspy_misc eds_hydrogenic_gos.HydrogenicGOS attribute), 478
quantification() (hyper-
spy_signals eds_tem.EDSTEM_mixin method), 318
Quantification () (in module hyperspy_signals eds_tem.EDSTEM_mixin method), 318

Index 731
read_raw() (in module hyperspy.io_plugins.ripple), 438
read_short() (in module hyperspy.misc.io.utils_readfile), 485
read_simple_data() (in module hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader), 412
read_string() (in module hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader), 412
read_struct() (in module hyperspy.io_plugins.digital_micrograph.DigitalMicrographReader), 412
read_ulong() (in module hyperspy.misc.io.utils_readfile), 485
read_ulong_long() (in module hyperspy.misc.io.utils_readfile), 485
read_u_short() (in module hyperspy.misc.io.utils_readfile), 486
readgosfile() (in module hyperspy.misc.eels.hartree_slater_gos.HartreeSlaterGOS), 477, 478
readLELong() (in module hyperspy.io_plugins.fei), 427
readLELongLong() (in module hyperspy.io_plugins.fei), 427
readLEShort() (in module hyperspy.io_plugins.fei), 427
real() (in module hyperspy.signals.complex_signal.ComplexSignal_mixin), 295
rebin() (in module hyperspy.signals.eds.EDS_mixin), 309
rebin() (in module hyperspy.signals.eels.EELSSpectrum_mixin), 329
rebin() (in module hyperspy.signals.lazy.LazySignal), 343
rebin() (in module hyperspy.signal.BaseSignal), 635
rebin() (in module hyperspy.misc.array_tools), 487
recompute_out() (in module hyperspy.interactive.Interactive), 580
reconstruct() (in module hyperspy.misc.holography.reconstruct), 482
reconstruct_component() (in module hyperspy.model), 598
reconstruct_object() (in module hyperspy.misc.export_dictionary), 492
reconstruct_phase() (in module hyperspy.signals.hologram_image.HologramImage), 332
record_by() (in module hyperspy.io_plugins.digital_micrograph.ImageObject), 412
Rectangle (class in module hyperspy.drawing._markers.rectangle), 374
RectangleWidget (class in module hyperspy.drawing._widgets.rectangles), 383
RectangularROI (in module hyperspy.roi), 606
red_chisq (in module hyperspy.model.BaseModel), 586
red_chisq (in module hyperspy.models.model1d.Model1D), 533
red_chisq (in module hyperspy.models.model2d.Model2D), 539
red_chisq_test (in module hyperspy.samfire_utils.goodness_of_fit_tests), 541
refresh() (in module hyperspy.samfire_utils.strategy.GlobalStrategy), 547
refresh() (in module hyperspy.samfire_utils.strategy.LocalStrategy), 548
refresh_database() (in module hyperspy.samfire.Samfire), 610, 612
regular_array2rgbx() (in module hyperspy.misc.rgb_tools), 503
rel_std_of_fraction() (in module hyperspy.misc.eels.tools), 480
release() (in module hyperspy.drawing._widgets.range.ModifiableSpanSelector), 381
remove() (in module hyperspy.axes.AxesManager), 564
remove() (in module hyperspy.drawing._widgets.scalebar.ScaleBar), 385
remove() (in module hyperspy.model.BaseModel), 586, 595
remove() (in module hyperspy.models.edsmodel.EDSMiddleware), 522
remove() (in module hyperspy.models.eelsmodel.EELSSpectrum_mixin), 529
remove() (in module hyperspy.models.model1d.Model1D), 534, 538
remove() (in module hyperspy.models.model2d.Model2D), 540
remove() (in module hyperspy.samfire.Samfire), 610, 612
remove() (in module hyperspy.samfire.StrategyList), 612
remove() (in module hyperspy.samfire_utils.strategy.SamfireStrategy), 549
remove() (in module hyperspy.signal.ModelManager), 656
remove_background() (in module hyperspy.signals.signal1d.Signal1D), 353
remove_EELS_edges_markers() (in module hyperspy.signals.eels.EELSSpectrum_mixin), 438
Index 735
sigma() (hyperspy_components.gaussianhf.GaussianHF property), 263
sigma_major (hyperspy_components.gaussian2d.Gaussian2D attribute), 259
sigma_major() (hyperspy_components.gaussian2d.Gaussian2D property), 261
sigma_minor (hyperspy_components.gaussian2d.Gaussian2D attribute), 259
sigma_minor() (hyperspy_components.gaussian2d.Gaussian2D property), 261
signal (hyperspy_model.BaseModel attribute), 586
signal (hyperspy.models.model1d.Model1D attribute), 533
signal (hyperspy.models.model2d.Model2D attribute), 539
signal() (hyperspy.models.model1d.Model1D property), 539
signal() (hyperspy.models.model2d.Model2D property), 541
Signal1D (class in hyperspy_signals.signal1d), 345
Signal1D() (hyperspy.models.eelmodel.EELSModel property), 530
Signal1DCalibration (class in hyperspy_signal_tools), 661
Signal1DFigure (class in hyperspy_drawing.signal1d), 390
Signal1DLine (class in hyperspy_drawing.signal1d), 391
Signal1DRangeSelector (class in hyperspy_signal_tools), 662
Signal2D (class in hyperspy_signals.signal2d), 358
signal_extent () (hyperspy.axes.AxesManager property), 564
signal_indices_in_array() (hyperspy_axes.AxesManager property), 564
signal_mask (hyperspy_learn.mva.LearningResults attribute), 447
signal_type () (hyperspy_io_plugins.digital_micrograph.ImageObject property), 413
SignalDimensionError, 580
signals (hyperspy_io_plugins.emd.EMD attribute), 421
SignalSizeError, 580
simple_decoration() (in module hyperspy_decorators), 574
simple_type (hyperspy_io_plugins.digital_micrograph.DigitalMicrographReader attribute), 412
SimpleMessage (class in hyperspy_signal_tools), 662
single_kernel() (in module hyperspy_samfire_utils.samfire_kernel), 543
size() (hyperspy_drawing.widget.ResizableDraggableWidgetBase property), 401
size() (hyperspy_io_plugins.digital_micrograph.ImageObject property), 413
skewness () (hyperspy_components.skew_normal.SkewNormal property), 284
SkewNormal (class in hyperspy_components.skew_normal), 282
skipif4() (hyperspy_io_plugins.digital_micrograph.DigitalMicrographReader method), 412
sleep() (hyperspy_samfire_utils.samfire_pool.SamfirePool method), 544
sleep() (hyperspy_samfire_utils.samfire_worker.Worker method), 546
sleep() (hyperspy_utilities.parallel_pool.ParallelPool method), 550, 552
slugify() (in module hyperspy_misc.utils), 515
smart_fit() (hyperspy_models.eelmodel.EELSModel method), 530
smooth_lowess() (hyperspy_signals.signal1d.Signal1D method), 354
smooth_savitzky_golay() (hyperspy_signals.signal1d.Signal1D method), 355
smooth_tv() (hyperspy_signals.signal1d.Signal1D method), 356
Smoothing (class in hyperspy_signal_tools), 662
SmoothingLowess (class in hyperspy_signal_tools), 662
SmoothingSavitzkyGolay (class in hyperspy_signal_tools), 663
SmoothingTV (class in hyperspy_signal_tools), 663
snap_all() (hyperspy_drawing.widget.ResizableDraggableWidgetBase property), 401
snap_position() (hyperspy_drawing.widget.DraggableWidgetBase property), 400
sobel_filter() (in module hyperspy_signal_tools), 365
span_selector_changed() (hyperspy_signal_tools.BackgroundRemoval method), 658
span_selector_changed() (hyperspy_signal_tools.SpikesRemoval method), 664
span_selector_switch() (hyperspy_signal_tools.ImageContrastEditor method), 660
HyperSpy Documentation, Release 1.6.0

unit_vector() (in module hyperspy.drawing._widgets.line2d), 381
update() (hyperspy.interactive.Interactive method), 580

UnitConversion (class in hyperspy.axes), 565
update() (hyperspy.roi.BaseInteractiveROI method), 600

units (hyperspy.io_plugins.semper_unf.SemperFormat attribute), 440
update() (hyperspy.roi.BaseROI method), 601

units() (hyperspy.axes.UnitConversion property), 566
update() (hyperspy.samfire.Samfire method), 610, 612

units() (hyperspy.io_plugins.digital_micrograph.ImageView property), 413
update() (hyperspy.samfire.utils.segmenters.histogram.HistogramSegmenter method), 542

units_factor() (hyperspy.models.edsmodel.EDSModel property), 522
update_active_edge() (hyperspy.signal_tools.EdgesRange method), 659

unmixing_matrix (hyperspy.learn.mva.LearningResults attribute), 447
update_axes_attributes_from() (hyperspy.axes.AxesManager method), 564

unpack_from_intbytes() (in module hyperspy.io_plugins.semper_unf), 441
update_axis() (hyperspy.axes.DataAxis method), 565

unpack_new_packed_complex() (hyperspy.io_plugins.digital_micrograph.ImageView method), 413
update_date_time_in_metadata() (in module hyperspy.misc.date_time_tools), 489

unpack_packed_complex() (hyperspy.io_plugins.digital_micrograph.ImageView method), 413
update_every (hyperspy.samfire.Samfire attribute), 610

unset_all_edges_intensities_positive() (hyperspy.models.eelsmodel.EELSModel method), 532
update_from() (hyperspy.axes.DataAxis method), 565

unwraped_phase() (hyperspy_signals.complex_signal.ComplexSignalMixin method), 295
update_histogram() (hyperspy.signal_tools.ImageContrastEditor method), 660

update_index_bounds() (hyperspy.axes.DataAxis method), 565
update_line() (hyperspy.signal_tools.EdgesRange method), 660

update_index_bounds() (hyperspy.axes.DataAxis method), 565
update_number_parameters() (hyperspy.component.Component method), 569

update_lines() (hyperspy.signal_tools.Smoothing method), 662
update_parameters() (hyperspy.samfire.utils.samfire_pool.SamfirePool method), 543, 545

update() (hyperspy.drawing._markers.horizontal_line.HorizontalLine method), 371
update_plot() (hyperspy.signal_tools.Smoothing method), 664

update() (hyperspy.drawing._markers.horizontal_line_segment.HorizontalLineSegment method), 372
update_plot() (hyperspy.model.BaseModel method), 586, 598

update() (hyperspy.drawing._markers.line_segment.LineSegment method), 372
update_plot() (hyperspy.models.model1d.Model1D method), 534

update() (hyperspy.drawing._markers.point.Point method), 373
update_plot() (hyperspy.signal.BaseSignal method), 535

update() (hyperspy.drawing._markers.rectangle.Rectangle method), 374
update_plot() (hyperspy.signal_tools.SpikesRemoval method), 664

update() (hyperspy.drawing._markers.text.Text method), 375
update_plot() (hyperspy.models.model1d.Model1D method), 375

update() (hyperspy.drawing._markers.vertical_line.VerticalLine method), 376
update_plot() (hyperspy.signal.Signal1D method), 381

update() (hyperspy.drawing.widgets.range.ModifiableSpanSelector method), 382
update_range() (hyperspy.drawing.widgets.range.ModifiableSpanSelector method), 381

update() (hyperspy.drawing.tiles.HistogramTilePlot method), 390
update_span_selector() (hyperspy.signal_tools.ImageContrastEditor method), 391
Voigt (class in hyperspy._components.voigt), 286
voigt() (in module hyperspy._components.pes_voigt), 273
VolumePlasmonDrude (class in hyperspy._components.volume_plasmon_drude), 289

W
weight() (hyperspy.samfire_utils.strategy.LocalStrategy property), 548
weight_to_atomic() (in module hyperspy.misc.material), 499
whiten_data() (in module hyperspy.learn.whitening), 468
Widget1DBase (class in hyperspy.drawing.widget), 403
Widget2DBase (class in hyperspy.drawing.widget), 403
WidgetBase (class in hyperspy.drawing.widget), 403
width() (hyperspy.drawing._widgets.rectangles.RectangleWidget property), 384
width() (hyperspy.roi.RectangularROI property), 607
Worker (class in hyperspy.samfire_utils.samfire_worker), 546
workers (hyperspy.samfire.Samfireattribute), 609
write_file() (hyperspy.io_plugins.emd.EMD_NCEM method), 423
write_raw() (in module hyperspy.io_plugins.ripple), 439
write_rpl() (in module hyperspy.io_plugins.ripple), 439
write_signal() (in module hyperspy.io_plugins.hspy), 428
WrongObjectError, 580

X
nan() (hyperspy.roi.RectangularROI property), 607
xc_template (hyperspy.signal_tools.PeaksFinder2D attribute), 661
xml2dtb() (in module hyperspy.misc.io.tools), 485
xray_lines_model() (in module hyperspy.misc.eds.utils), 473
xray_range() (in module hyperspy.misc.eds.utils), 473

Y
y() (hyperspy.roi.RectangularROI property), 607

Z
zeta_to_edx_cross_section() (in module hyperspy.misc.eds.utils), 474