

---

# **LumiSpy**

*Release 0.2.2*

**The LumiSpy developers**

**Aug 08, 2023**



# USER GUIDE

<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Introduction . . . . .	4
1.3	Non-uniform signal axes . . . . .	7
1.4	Signal tools . . . . .	9
1.5	Fitting luminescence data . . . . .	11
1.6	Utility functions . . . . .	12
1.7	LumiSpy metadata structure . . . . .	13
1.8	Bibliography . . . . .	18
1.9	lumispy . . . . .	18
1.10	Changelog . . . . .	34
1.11	Citing LumiSpy . . . . .	36
1.12	Contributing . . . . .	36
1.13	License . . . . .	38
	<b>Bibliography</b>	<b>39</b>
	<b>Python Module Index</b>	<b>41</b>
	<b>Index</b>	<b>43</b>



**LumiSpy** is a Python package extending the functionality for multi-dimensional data analysis provided by the [HyperSpy](#) library. It is aimed at helping with the analysis of luminescence spectroscopy data (cathodoluminescence, photoluminescence, electroluminescence, Raman, SNOM).

Check out the [Installation](#) section for further information, including how to start using this project.

Complementing this documentation, the [LumiSpy Demos](#) repository contains curated Jupyter notebooks to provide tutorials and exemplary workflows.

---

**Note:** This project is under active development. Everyone is welcome to contribute. Please read our (see [Contributing](#)) guidelines and get started!

---



## CONTENTS

### 1.1 Installation

To install LumiSpy, you have the following options (independent of the operating system you use):

1. LumiSpy is included in the [HyperSpy Bundle](#), a standalone program that includes a python distribution and all relevant libraries (recommended if you do not use *python* for anything else).
2. *Installation using conda* (recommended if you are also working with other *python* packages).
3. *Installation using pip*.
4. Installing the development version from [GitHub](#). Refer to the appropriate section in the [HyperSpy user guide](#) (replacing `hyperspy` by `lumispy`).

#### 1.1.1 Installation using conda

Follow these 3 steps to install LumiSpy using **conda** and start using it.

##### 1. Creating a conda environment

LumiSpy requires Python 3 and conda – we suggest using the Python 3 version of [Miniconda](#).

We recommend creating a new environment for the LumiSpy package (or installing it in the [HyperSpy](#) environment, if you have one already). To create a new environment:

1. Load the anaconda prompt.
2. Run the following command:

```
(base) conda create -n lumispy -y
```

##### 2. Installing the package in the new environment

Now activate the LumiSpy environment and install the package from `conda-forge`:

```
(base) conda activate lumispy  
(lumispy) conda install -c conda-forge lumispy -y
```

Required dependencies will be installed automatically.

Installation is completed! To start using it, check the next section.

**Note:** If you run into trouble, check the more detailed documentation in the [HyperSpy user guide](#).

---

### 3. Getting Started

To get started using LumiSpy, especially if you are unfamiliar with Python, we recommend using [Jupyter notebooks](#). Having installed LumiSpy as above, a Jupyter notebook can be installed and opened using the following commands entered into an anaconda prompt (from scratch):

```
(base) conda activate lumispy
(lumispy) conda install -c conda-forge jupyterlab -y
(lumispy) jupyter lab
```

[Tutorials and exemplary workflows](#) have been curated as a series of Jupyter notebooks that you can work through and modify to perform many common analyses.

#### 1.1.2 Installation using pip

Alternatively, you can also find LumiSpy in the [Python Package Index \(PyPI\)](#) and install it using (requires pip):

```
pip install lumispy
```

Required dependencies will be installed automatically.

#### 1.1.3 Updating the package

Using **conda**:

```
conda update lumispy -c conda-forge
```

Using **pip**:

```
pip install lumispy --upgrade
```

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

---

## 1.2 Introduction

### 1.2.1 What is LumiSpy

**LumiSpy** is an open-source python library aimed at helping with the analysis of luminescence spectroscopy data – the development started mainly with photoluminescence (PL), cathodoluminescence (CL), electroluminescence (EL) and Raman spectroscopy in mind. Besides the standard continuous-excitation spectral data, the idea is to provide tools also for the analysis of time-resolved (transient) measurements. However, it may prove useful also for other optical measurements, such as absorption or transmission spectroscopy, scanning optical near field microscopy (SNOM), as well as fourier-transform infrared spectroscopy (FTIR).

**LumiSpy** is an extension to the python package **HyperSpy** that facilitates hyperspectral data analysis, i.e. maps or linescans where a spectrum is collected at each pixel. Or more general, multidimensional datasets that can be described as multidimensional arrays of a given signal.

Notable features that **HyperSpy** provides are:

- **base signal classes** for the handling of (multidimensional) spectral data,
- the necessary tools for loading **various data file formats**,
- **analytical tools** that exploit the multidimensionality of datasets,
- a user-friendly and powerful framework for **model fitting** that provides many standard functions and can be easily extended to custom ones,
- **machine learning** algorithms that can be useful, e.g. for denoising data,
- efficient handling of **big datasets**,
- functions for **data visualization** both to evaluate datasets during the analysis and provide interactive operation for certain functions, as well as for plotting of data,
- extracting subsets of data from multidimensional datasets via **regions of interest** and a powerful numpy-style **indexing mechanism**,
- handling of **non-uniform data axes** (introduced in the v1.7 release).

**LumiSpy** provides in particular:

- additional *Signal types* specifically for luminescence spectra and transients,
- transformation to *non-uniform signal axes* for use of other common units, such as eV (electron volt) and wavenumbers (Raman shift),
- additional *signal tools* such as data normalization and scaling,
- various *utility functions* useful in luminescence spectroscopy data analysis, such as joining multiple spectra along the signal axis, unit conversion, etc.

**LumiSpy** should facilitate an easy and reproducible analysis of single spectra or spectral images.

## 1.2.2 Signal types

As an extension to HyperSpy, LumiSpy provides several signal types extending the **base classes available in HyperSpy**. When the LumiSpy library is installed, these additional signal types are directly available to HyperSpy. To print all available specialised `hyperspy.signal.BaseSignal` subclasses installed in your system call the `hyperspy.utils.print_known_signal_types()` function:

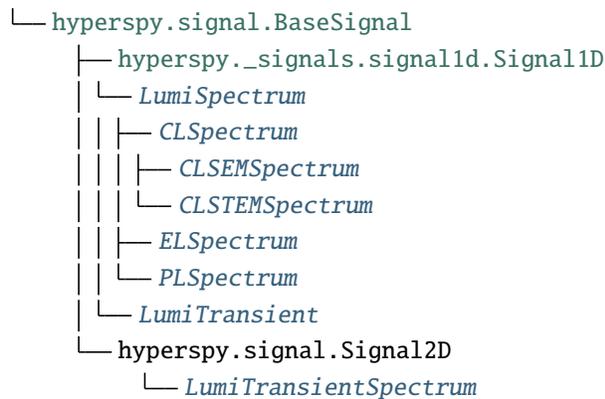
```
>>> import hyperspy.api as hs
>>> hs.print_known_signal_types()
```

The different subclasses are characterized by the `signal_type` metadata attribute. Some additional properties are summarized in the table below. Depending on the use case, certain functions will only be available for some signal types (or inheriting) signal types.

Table 1: LumiSpy subclasses and their basic attributes.

BaseSignal sub-class	sig- nal_dimensio	sig- nal_type	dtype	aliases
<i>LumiSpectrum</i>	1	Lumines- cence	real	LumiSpectrum, LuminescenceSpectrum
<i>CLSpectrum</i>	1	CL	real	CLSpectrum, cathodoluminescence
<i>CLSEMSpectrum</i>	1	CL_SEM	real	CLSEM, cathodoluminescence SEM
<i>CLSTEMSpectrum</i>	1	CL_STEM	real	CLSTEM, cathodoluminescence STEM
<i>ELSpectrum</i>	1	EL	real	ELSpectrum, electroluminescence
<i>PLSpectrum</i>	1	PL	real	PLSpectrum, photoluminescence
<i>LumiTransient</i>	1	Transient	real	TRLumi, TR luminescence, time-resolved luminescence
<i>LumiTransientSpec</i>	2	Tran- sientSpec	real	TRLumiSpec, TR luminescence spectrum, time-resolved luminescence spectrum

The hierarchy of the LumiSpy signal types and their inheritance from HyperSpy is summarized in the following diagram:



### 1.2.3 Where are we heading?

LumiSpy is under active development, and as a user-driven project, we welcome contributions (see [Contributing](#)) to the code and documentation, but also bug reports and feature requests from any other users. Don't hesitate to join the discussions!

Currently, we have implemented the base functionality that extends [HyperSpy's capabilities](#) to additional signal classes. In the near future, the following functions should be developed:

- handling of transient (time-resolved) data,
- reading of common PL data formats,
- more dedicated analysis functionalities,
- ...

## 1.3 Non-uniform signal axes

LumiSpy facilitates the use of **non-uniform axes**, where the points of the axis vector are not uniformly spaced. This situation occurs in particular when converting a wavelength scale to energy (eV) or wavenumbers (e.g. for Raman shifts).

The conversion of the signal axis can be performed using the functions `to_eV()`, `to_invcm()` and `to_raman_shift()` (alias for `to_invcm_relative()`). If the unit of the signal axis is set, the functions can handle wavelengths in either nm or  $\mu\text{m}$ .

Accepted parameters are `inplace=True/False` (default is True), which determines whether the current signal object is modified or a new one is created, and `jacobian=True/False` (default is True, see *Jacobian transformation*).

### 1.3.1 The energy axis

The transformation from wavelength  $\lambda$  to energy  $E$  is defined as  $E = hc/\lambda$ . Taking into account the refractive index of air and doing a conversion from nm to eV, we get:

$$E[\text{eV}] = \frac{10^9}{e} \frac{hc}{n_{\text{air}}\lambda[\text{nm}]},$$

where  $h$  is the Planck constant,  $c$  is the speed of light,  $e$  is the elementary charge and  $n_{\text{air}}$  is the refractive index of air, see also [Pfueller].

```
>>> s2 = s.to_eV(inplace=False)
>>> s.to_eV()
```

**Note:** The refractive index of air  $n_{\text{air}}$  is wavelength dependent. This dependence is taken into account by LumiSpy based on the analytical formula given by [Peck] valid from 185-1700 nm (outside of this range, the values of the refractive index at the edges of the range are used and a warning is raised).

### 1.3.2 The wavenumber axis/Raman shifts

The transformation from wavelength  $\lambda$  to wavenumber  $\tilde{\nu}$  (spatial frequency of the wave) is defined as  $\tilde{\nu} = 1/\lambda$ . The wavenumber is usually given in units of  $\text{cm}^{-1}$ .

When converting a signal to Raman shift, i.e. the shift in wavenumbers from the exciting laser wavelength, the laser wavelength has to be passed to the function using the parameter `laser` using the same units as for the original axis (e.g. 325 for nm or 0.325 for  $\mu\text{m}$ ) unless it is contained in the *LumiSpy metadata structure* under `Acquisition_instrument.Laser.wavelength`.

TODO: Automatically read laser wavelength from metadata if given there.

```
>>> s2 = s.to_invcm(inplace=False)
>>> s.to_invcm()
>>> s2 = s.to_raman_shift(inplace=False, laser=325)
>>> s.to_raman_shift(laser=325)
```

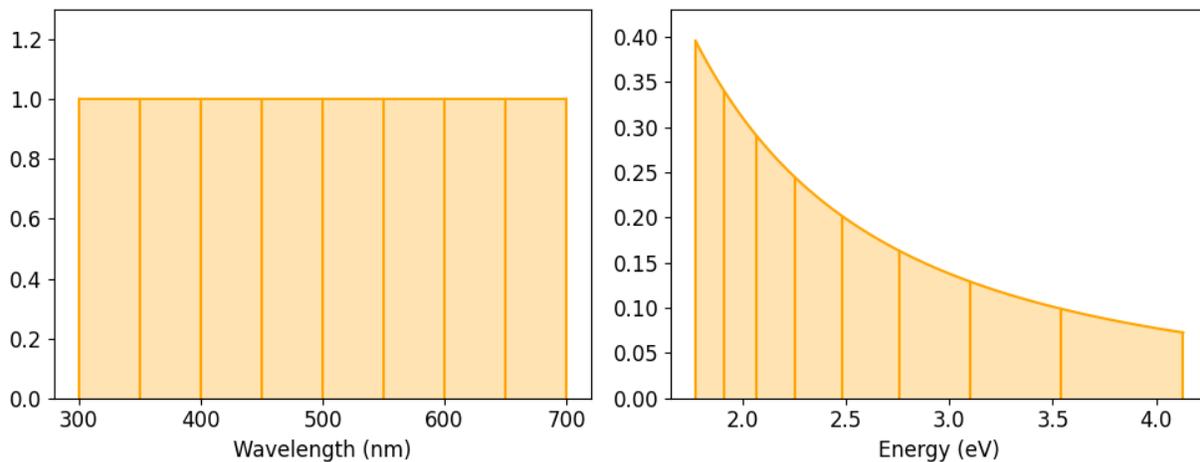
### 1.3.3 Jacobian transformation

When transforming the signal axis, the signal intensity is automatically rescaled (Jacobian transformation), unless the `jacobian=False` option is given. Only converting the signal axis, and leaving the signal intensity unchanged, implies that the integral of the signal over the same interval would lead to different results depending on the quantity on the axis (see e.g. [Mooney] and [Wang]).

For the energy axis as example, if we require  $I(E)dE = I(\lambda)d\lambda$ , then  $E = hc/\lambda$  implies

$$I(E) = I(\lambda) \frac{d\lambda}{dE} = I(\lambda) \frac{d}{dE} \frac{hc}{E} = -I(\lambda) \frac{hc}{E^2}$$

The minus sign just reflects the different directions of integration in the wavelength and energy domains. The same argument holds for the conversion from wavelength to wavenumber (just without the additional prefactors in the equation). The renormalization in LumiSpy is defined such that the intensity is converted from counts/nm (or counts/ $\mu\text{m}$ ) to counts/meV. The following figure illustrates the effect of the Jacobian transformation:



#### Transformation of the variance

Scaling the signal intensities implies that also the stored variance of the signal needs to be scaled accordingly. According to  $Var(aX) = a^2Var(X)$ , the variance has to be multiplied with the square of the Jacobian. This squared renormalization is automatically performed by LumiSpy if `jacobian=True`. In particular, homoscedastic (constant) noise will consequently become heteroscedastic (changing as a function of the signal axis vector). Therefore, if the `metadata.Signal.Noise_properties.variance` attribute is a constant, it is converted into a `hyperspy.signal.BaseSignal` object before the transformation.

See [Signal variance \(noise\)](#) for more general information on data variance in the context of model fitting and the HyperSpy documentation on [setting the noise properties](#).

---

**Note:** If the Jacobian transformation is performed, the values of `metadata.Signal.Noise_properties.Variance_linear_model` are reset to their default values (`gain_factor=1`, `gain_offset=0` and `correlation_factor=1`). Should these values deviate from the defaults, make sure to run `hyperspy.signal.BaseSignal.estimate_poissonian_noise_variance()` prior to the transformation.

---

## 1.4 Signal tools

This section summarizes functions operating on the signal data. Besides those implemented in LumiSpy, it highlights functions from HyperSpy that are particularly useful for luminescence spectroscopy data.

### 1.4.1 Scaling and normalizing signal data

For comparative plotting or a detailed analysis, the intensity of spectra may need to be either scaled by the respective integration times or normalized. The luminescence signal classes provide these functionalities in the methods `scale_by_exposure()` and `normalize()`.

Both functions can operate directly on the signal (`inplace=True`), but as default a new signal is returned.

The `scaling` function can use the `integration_time` (unit: seconds) provided in the *LumiSpy metadata structure* (`metadata.Acquisition_instrument.Detector.integration_time`). Otherwise, the appropriate parameter has to be passed to the function.

```
>>> scaled = s.scale_by_exposure(integration_time=0.5, inplace=True)
```

**Normalization** is performed for the pixel with maximum intensity. Alternatively, the parameter `pos` in calibrated units of the signal axis can be given to normalize the intensity at this position. Normalization may be convenient for plotting, but should usually not be performed on signals used as input for further analysis (therefore the default is `inplace=False`).

```
>>> s.normalize(pos=450)
```

### 1.4.2 Peak positions and properties

#### Peak identification

HyperSpy provides functions to find the positions of maxima or minima in a dataset:

- `indexmax()` - return the index of the maximum value along a given axis.
- `indexmin()` - return the index of the minimum value along a given axis.
- `valuemax()` - return the position/coordinates of the maximum value along a given axis in calibrated units.
- `valuemin()` - return the position/coordinates of the minimum value along a given axis in calibrated units.

These functions take the `axis` keyword to define along which axis to perform the operation and return a new signal containing the result.

A much more powerful method to identify peaks is using the **peak finding routine** based on the downward zero-crossings of the first derivative of a signal: `find_peaks1D_ohaver()`. This function can find multiple peaks in a dataset and has a number of parameters for fine-tuning the sensitivity, etc.

All of these functions can be performed for a subset of the dataset:

```
>>> peaks = s.find_peaks1D_ohaver()
>>> peaks = s.isig[100:-100].find_peaks1D_ohaver()
```

## Peak Width

For asymmetric peaks, *fitted functions* may not provide an accurate description of the peak, in particular the peak width. The function `estimate_peak_width()` determines the **width of a peak** at a certain fraction of its maximum value. The default value `factor=0.5` returns the full width at half maximum (FWHM).

```
>>> s.remove_background()
>>> width = s.estimate_peak_width(factor=0.3)
```

## Calculating the centroid of a spectrum (centre of mass)

The function `centroid()` (based on the utility function `com()`) is an alternative to finding the position of the maximum intensity of a peak, useful in particular for non-symmetric peaks with pronounced shoulders. It finds the centroid (center of mass) of a peak in the spectrum from the signal axis units (or pixel number) and the intensity at each pixel value. It basically represents a “weighted average” of the peak as such:

$$com = \frac{\sum x_i I_i}{\sum I_i},$$

where  $x_i$  is the wavelength (or pixel number) at which the intensity of the spectrum  $I_i$  is measured.

This function also works for non-linear axes. For the `hyperspy.axes.FunctionalDataAxis`, the centroid is extrapolated based on the function used to create the non-uniform axis. For `hyperspy.axes.DataAxis`, a linear interpolation between the axes points at the center of mass is assumed, but this behaviour can be changed with the *kwargs* of `scipy.interpolate.interp1d` function.

```
>>> s = lum.signals.LumiSpectrum([[1, 2, 3, 2, 1, 0]]*2)*3)
>>> s
<LumiSpectrum, title: , dimensions: (2, 3|6)>

>>> ax = s.axes_manager.signal_axes[0]
>>> ax.offset = 200
>>> ax.scale = 100

>>> com = s.centroid()
>>> com
<Signal2D, title: Centroid map, dimensions: (|2, 3)>
>>> com.data[0,0]
400.0
```

---

**Note:** This function only works for a single peak. If you have multiple peaks, slice the signal beforehand or use the slice parameter (which follows the `s.isig[:]` convention).

---

**Note:** The *Jacobian transformation* may affect the shape, in particular of broader peaks. It is therefore highly recommended to convert luminescence spectra from wavelength to the *energy axis* prior to determining the centroid to determine the true emission energy. See e.g. [Wang] and [Mooney].

---

### 1.4.3 Signal statistics and analytical operations

**Standard statistical operations** can be performed on the data or a subset of the data, notably these include `max()`, `min()`, `sum()`, `mean()`, `std()`, and `var()`. Variations of all these functions exist that ignore missing values (NaN) if present, e.g. `nanmax()`.

**Integration** along a specified signal axis is performed using the function `integrate1D()`.

The numerical **derivative** of a signal can be calculated using the function `derivative()`, while the  $n$ -th order **discrete difference** can be calculated using `diff()`.

These functions take the `axis` keyword to define along which axis to perform the operation and return a new signal containing the result:

```
>>> area = s.integrate1D(axis=0)
```

### 1.4.4 Replacing negative data values

Log-scale plotting fails in the presence of negative values in the dataset (e.g. introduced after background removal). In this case, the utility function `remove_negative()` replaces all negative values in the data array by a `basevalue` (default `basevalue=1`). The default operational mode is `inplace=False` (a new signal object is returned).

```
>>> s.remove_negative(0.1)
```

## 1.5 Fitting luminescence data

LumiSpy is compatible with [HyperSpy model fitting](#). It can fit using both uniform and non-uniform axes (e.g. energy scale). A general introduction can be found in the [HyperSpy user guide](#).

A detailed example is given in the [Fitting\\_tutorial](#) in the [HyperSpy demos repository](#). See also the [LumiSpy demo notebooks](#) for examples of data fitting.

---

**Note:** The *Jacobian transformation* may affect the shape, in particular of broader peaks. It is therefore highly recommended to convert luminescence spectra from wavelength to the *energy axis* prior to any fitting to obtain the true emission energy. See e.g. [\[Wang\]](#) and [\[Mooney\]](#).

---

TODO: Show how to extract the *modeled signal* with all/one component.

### 1.5.1 Signal variance (noise)

TODO: Documentation on variance handling in the context of fitting, in particular using `estimate_poissonian_noise_variance()`

For a detailed discussion, see [\[Tappy\]](#)

## 1.6 Utility functions

This section summarizes various useful functions implemented in LumiSpy.

### 1.6.1 Join spectra

In case several spectra (or spectral images) were subsequently recorded for different, but overlapping spectral windows, LumiSpy provides a utility `join_spectra()` to merge these into a single spectrum. The main argument is a list of two or more spectral objects. Spectra are joined at the centre of the overlapping range along the signal axis. To avoid steps in the intensity, several parameters (see docstring: `join_spectra()`) allow to tune the scaling of the later signals with respect to the previous ones. By default, the scaling parameter is determined as average ratio between the two signals in the range of +/- 50 pixels around the centre of the overlapping region.

```
>>> import lumispy as lum
>>> s = lum.join_spectra((s1,s2))
```

### 1.6.2 Utilities for spectral maps

The function `crop_edges()` removes the specified number of pixels from all four edges of a spectral map. It is a convenience wrapper for the `inav` method in `HyperSpy`.

```
>>> s.crop_edges(crop_px=2)
```

*[TODO: add possibility to crop different amounts of pixels on different sides]*

### 1.6.3 Unit conversion

For convenience, LumiSpy provides functions that convert between different units commonly used for the signal axis. Namely,

- `nm2eV()`
- `eV2nm()`
- `nm2invcm()`
- `invcm2nm()`

For the energy axis, the conversion uses the wavelength-dependent refractive index of air.

### 1.6.4 Solving the grating equation

The function `solve_grating_equation()` follows the conventions described in the tutorial from [Horiba Scientific](#).

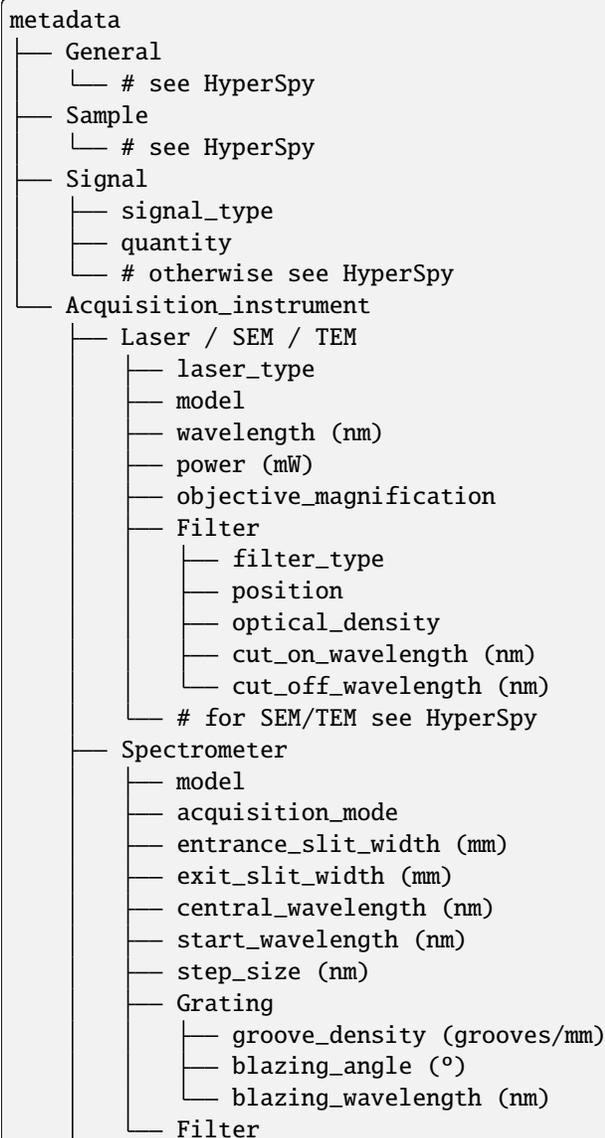
## 1.7 LumiSpy metadata structure

LumiSpy extends the [HyperSpy metadata structure](#) with conventions for metadata specific to its signal types. Refer to the [HyperSpy metadata documentation](#) for general metadata fields.

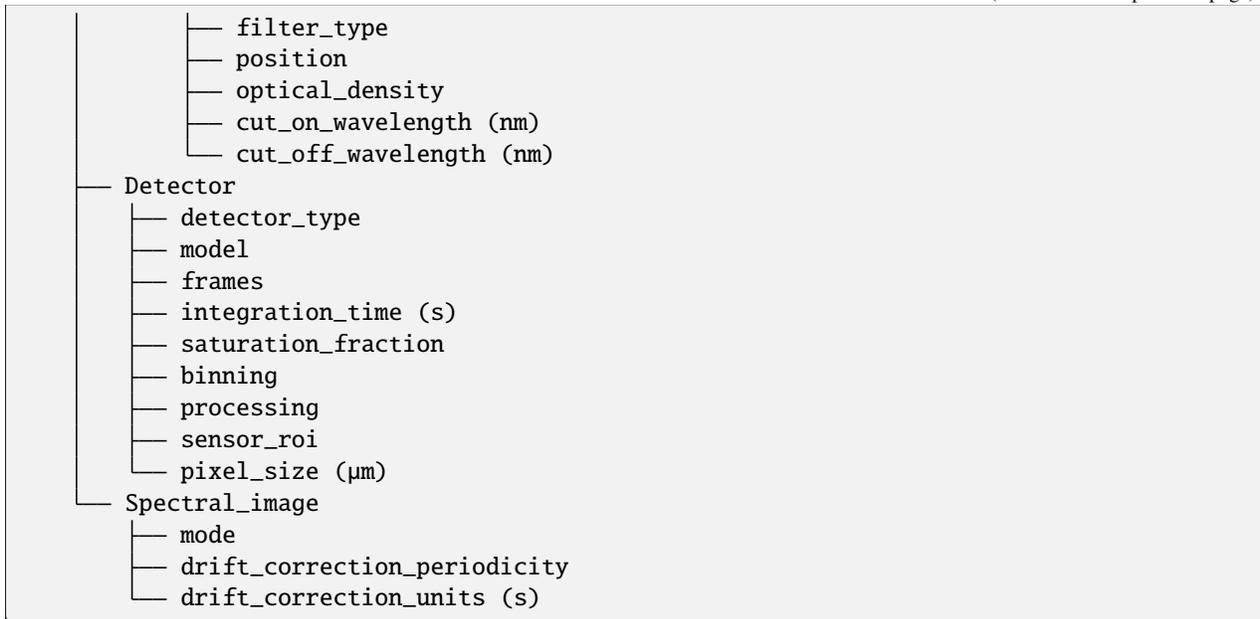
The metadata of any **signal objects** is stored in the *metadata* attribute, which has a tree structure. By convention, the node labels are capitalized and the ones for leaves are not capitalized. When a leaf contains a quantity that is not dimensionless, the units can be given in an extra leaf with the same label followed by the `_units` suffix.

Besides directly accessing the metadata tree structure, e.g. `s.metadata.Signal.signal_type`, the HyperSpy methods `set_item()`, `has_item()` and `get_item()` can be used to add to, search for and read from items in the metadata tree, respectively.

The luminescence specific 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. Note that not all types of leaves will apply to every type of measurement. For example, while parallel acquisition with a CCD is characterized by the `central_wavelength`, a serial acquisition with a PMT will require a `start_wavelength` and a `step_size`.



(continues on next page)



### 1.7.1 General

See [HyperSpy-Metadata-General](#).

### 1.7.2 Sample

See [HyperSpy-Metadata-Sample](#).

### 1.7.3 Signal

#### signal\_type

type: string

String that describes the type of signal. The LumiSpy specific signal classes are summarized under *Signal types*.

#### quantity

type: string

The name of the quantity of the “intensity axis” with the units in round brackets if required, for example ‘Intensity (counts/s)’.

See [HyperSpy-Metadata-Signal](#) for additional fields.

---

## 1.7.4 Acquisition Instrument

### 1.7.5 Laser / SEM / TEM

For **SEM** or **TEM** see [HyperSpy-Metadata-SEM](#) or [HyperSpy-Metadata-TEM](#).

#### Laser

**laser\_type**

type: string

The type of laser used, e.g. 'HeCd'.

**model**

type: string

Model of the laser (branding by manufacturer).

**wavelength**

type: float

Emission wavelength of the exciting laser in nm.

**power**

type: float

Measured power of the excitation laser in mW.

**magnification**

type: int

Magnification of the microscope objective used to focus the beam to the sample.

#### Filter

Information about additional filters entered into the lightpath before the sample. In case multiple filters are used, they should be numbered *Filter\_1*, etc.

**filter\_type**

type: string

Type of filter (e.g. 'optical density', 'short pass', 'long pass', 'bandpass', 'color').

**position**

type: string

Position in the beam (e.g. 'excitation' vs. 'detection' in case of optical excitation).

**optical\_density**

type: float

Optical density in case of an intensity filter.

**cut\_on\_wavelength**

type: float

Cut on wavelength in nm in case of a long-pass or bandpass filter.

**cut\_off\_wavelength**

type: float

Cut off wavelength in nm in case of a short-pass or bandpass filter.

### 1.7.6 Spectrometer

Contains information about the spectrometer, configuration and grating used for the measurement. In case multiple spectrometers are connected in series, they should be numbered *Spectrometer\_1*, etc.

**model**

type: string

Model of the spectrometer (branding by manufacturer).

**acquisition\_mode**

type: string

Acquisition mode (e.g. 'Parallel dispersive', versus 'Serial dispersive').

**entrance\_slit\_width**

type: float

Width of the entrance slit in mm.

**exit\_slit\_width**

type: float

Width of the exit slit (serial acquisition) in mm.

**central\_wavelength**

type: float

Central wavelength during acquisition (parallel acquisition).

**start\_wavelength**

type: float

Start wavelength in nm (serial acquisition).

**step\_size**

type: float

Step size in nm (serial acquisition).

### Grating

Information of the dispersion grating employed in the measurement.

**groove\_density**

type: int

Density of lines on the grating in grooves/mm.

**blazing\_angle**

type: int

Angle in degree (°) that the grating is blazed at.

**blazing\_wavelength**

type: int

Wavelength that the grating blaze is optimized for in nm.

## Filter

Information about additional filters entered into the lightpath after the sample. In case multiple filters are used, they should be numbered *Filter\_1*, etc. See *Filter* above for details on items that may potentially be included.

### 1.7.7 Detector

Contains information about the detector used to acquire the signal. Contained leaves will differ depending on the type of detector.

#### **detector\_type**

type: string

The type of detector used to acquire the signal (CCD, PMT, StreakCamera, TCSPD).

#### **model**

type: string

The model of the used detector.

#### **frames**

type: int

Number of frames that are summed to yield the total integration time.

#### **integration\_time (s)**

type: float

Time over which the signal is integrated. In case multiple frames are summed, it is the total exposure time. In case of serial acquisition, it is the dwell time per data point.

#### **saturation\_fraction**

type: float

Fraction of the signal intensity compared with the saturation threshold of the CCD.

#### **binning**

type: tuple of int

A tuple that describes the binning of a parallel detector such a CCD on readout in x and y directions.

#### **processing**

type: string

Information about automatic processing performed on the data, e.g. 'dark subtracted'.

#### **sensor\_roi**

type: tuple of int

Tuple of length 2 or 4 that specifies range of pixels on a detector that are read out: (offset x, offset y, size x, size y) for a 2D array detector and (offset, size) for a 1D line detector.

#### **pixel\_size**

type: float or tuple of float

Size of a pixel in  $\mu\text{m}$ . Tuple of length 2 (width, height), when the pixel is not square.

## 1.7.8 Spectral\_image

Contains information about mapping parameters, such as step size, drift correction, etc.

### **mode**

type: string

Mode of the spectrum image acquisition such as 'Map' or 'Linescan'.

### **drift\_correction\_periodicity**

type: int/float

Periodicity of the drift correction in specified units (standard s).

### **drift\_correction\_units**

type: string

Units of the drift correction such as 's', 'px', 'rows'.

## 1.8 Bibliography

## 1.9 lumispy

### 1.9.1 lumispy package

#### Subpackages

#### lumispy.signals package

#### Submodules

#### lumispy.signals.cl\_spectrum module

#### Signal class for cathodoluminescence spectral data

```
class lumispy.signals.cl_spectrum.CLSEMSpectrum(*args, **kwargs)
```

Bases: *CLSpectrum*

**1D scanning electron microscopy cathodoluminescence signal class.**

```
correct_grating_shift(cal_factor_x_axis, corr_factor_grating, sem_magnification, **kwargs)
```

Applies shift caused by the grating offset wrt the scanning centre. Authorship: Gunnar Kusch (gk419@cam.ac.uk)

#### Parameters

- **cal\_factor\_x\_axis** – The navigation correction factor.
- **corr\_factor\_grating** – The grating correction factor.
- **sem\_magnification** – The SEM (real) magnification value. For the Attolight original metadata, take the *SEM.Real\_Magnification* value
- **kwargs** – The parameters passed to *hyperspy.align1D()* function like: interpolation\_method ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic') parallel: Bool crop, expand, fill\_value ...

```
class lumispy.signals.cl_spectrum.CLSTEMSpectrum(*args, **kwargs)
```

Bases: *CLSpectrum*

**1D scanning transmission electron microscopy cathodoluminescence signal class.**

```
class lumispy.signals.cl_spectrum.CLSpectrum(*args, **kwargs)
```

Bases: *LumiSpectrum*

**General 1D cathodoluminescence signal class.**

```
_make_signal_mask(luminescence_roi)
```

Creates a mask from the peak position and peak widths of the luminescence spectrum.

**Parameters**

**luminescence\_roi** (*array*) – In the form of an array of pairwise elements [[peak1\_x, peak1\_width], [peak2\_x, peak2\_width],...].

**Returns**

A *signal\_mask*.

**Return type**

*array*

```
remove_spikes(threshold='auto', show_diagnosis_histogram=False, inplace=False,
               luminescence_roi=None, signal_mask=None, add_noise=False, navigation_mask=None,
               interactive=False, **kwargs)
```

HyperSpy-based spike removal tool adapted to LumiSpy to run non-interactively and without noise addition by default. Graphical interface to remove spikes from EELS spectra or luminescence data. If non-interactive, it removes all spikes and returns a *~hyperspy.signals.\_signal\_tools.SpikesRemoval* object.

**Parameters**

- **signal\_mask** (*bool array*) – Restricts the operation to the signal locations not marked as True (masked).
- **navigation\_mask** (*bool array*) – Restricts the operation to the navigation locations not marked as True (masked).
- **threshold** (*'auto' or int*) – if *int* set the threshold value use for the detecting the spikes. If *auto*, determine the threshold value as being the first zero value in the histogram obtained from the `spikes_diagnosis()` method.
- **interactive** (*bool*) – If True, remove the spikes using the graphical user interface. If False, remove all the spikes automatically, which can introduce artefacts if used with signal containing peak-like features. However, this can be mitigated by using the *signal\_mask* argument to mask the signal of interest.
- **display** (*bool*) – If True, display the user interface widgets. If False, return the widgets container in a dictionary, usually for customisation or testing.
- **toolkit** (*str, iterable of 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*) –

**Keyword arguments pass to**  
`SpikesRemoval()`

**See also:**

`spikes_diagnosis()`

**Parameters**

- **show\_diagnosis\_histogram** (*bool*) – Plot or not the derivative histogram to show the magnitude of the spikes present.
- **inplace** (*bool*) – If False, a new signal object is created and returned. If True, the original signal object is modified.
- **luminescence\_roi** (*array*) – The peak position and peak widths of the peaks in the luminescence spectrum. In the form of an array of pairwise elements [[peak1\_x, peak1\_width], [peak2\_x, peak2\_width],...] in the units of the signal axis. It creates a *signal\_mask* protecting the peak regions. To be used instead of *signal\_mask*.

**Returns**

Depends on *inplace*, returns or overwrites the *CLSpectrum* after spike removal.

**Return type**

None or *CLSpectrum*

```
class lumispy.signals.cl_spectrum.LazyCLSEMSpectrum(*args, **kwargs)
```

Bases: *LazySignal*, *CLSEMSpectrum*

**Lazy 1D scanning electron microscopy cathodoluminescence signal class.**

```
class lumispy.signals.cl_spectrum.LazyCLSTEMSpectrum(*args, **kwargs)
```

Bases: *LazySignal*, *CLSTEMSpectrum*

**Lazy 1D scanning transmission electron microscopy cathodoluminescence signal class.**

```
class lumispy.signals.cl_spectrum.LazyCLSpectrum(*args, **kwargs)
```

Bases: *LazySignal*, *CLSpectrum*

**General lazy 1D cathodoluminescence signal class.**

**lumispy.signals.common\_luminescence module****Signal class for luminescence data (BaseSignal class)**

```
class lumispy.signals.common_luminescence.CommonLumi
```

Bases: *object*

**General luminescence signal class (dimensionless)**

```
crop_edges(crop_px)
```

Crop the amount of pixels from the four edges of the scanning region, from out the edges inwards.

**Parameters**

**crop\_px** (*int*) – Amount of pixels to be cropped on each side individually.

**Returns**

**signal\_cropped** – A smaller cropped CL signal object. If *inplace* is True, the original object is modified and no *LumiSpectrum* is returned.

**Return type**

*CommonLuminescence*

**normalize**(*pos=nan, element\_wise=False, inplace=False*)

Normalizes data to value at *pos* along signal axis, defaults to maximum value.

Can be helpful for e.g. plotting, but does not make sense to use on signals that will be used as input for further calculations!

#### Parameters

- **pos** (*float, int*) – If ‘nan’ (default), spectra are normalized to the maximum. If *float*, position along signal axis in calibrated units at which to normalize the spectra. If *int*, index along signal axis at which to normalize the spectra.
- **element\_wise** (*boolean*) – If *False* (default), a spectrum image is normalized by a common factor. If *True*, each spectrum is normalized individually.
- **inplace** (*boolean*) – If *False* (default), a new signal object is created and returned. If *True*, the operation is performed on the existing signal object.

#### Notes

Sets *metadata.Signal.normalized* to *True*. If *metadata.Signal.quantity* contains the word ‘Intensity’, replaces this field with ‘Normalized intensity’.

**remove\_negative**(*basevalue=1, inplace=False*)

Sets all negative values to ‘basevalue’, e.g. for logarithmic scale plots.

#### Parameters

- **basevalue** (*float*) – Value by which negative values are replaced (default = 1).
- **inplace** (*boolean*) – If *False* (default), a new signal object is created and returned. Otherwise, the operation is performed on the existing signal object.

#### Notes

Sets *metadata.Signal.negative\_removed* to *True*.

**scale\_by\_exposure**(*integration\_time=None, inplace=False, \*\*kwargs*)

Scale data in spectrum by integration time / exposure, (e.g. convert counts to counts/s).

#### Parameters

- **integration\_time** (*float*) – Integration time (exposure) in s. If not given, the function tries to use the ‘*metadata.Acquisition\_instrument.Detector.integration\_time*’ field or alternatively find any ‘*integration\_time*’, ‘*exposure*’ or ‘*dwell\_time*’ fields in the *original\_metadata*.
- **inplace** (*boolean*) – If *False* (default), a new signal object is created and returned. If *True*, the operation is performed on the existing signal object.

## Notes

Sets `metadata.Signal.scaled` to `True`. If intensity units is 'counts', replaces them by 'counts/s'.

Deprecated since version 0.2: The `exposure` argument was renamed `integration_time`, and it will be removed in LumiSpy 1.0.

## lumispypy.signals.common\_transient module

### Signal class for transient data (BaseSignal class)

```
class lumispypy.signals.common_transient.CommonTransient
```

Bases: `object`

General transient signal class (dimensionless)

## lumispypy.signals.el\_spectrum module

### Signal class for electroluminescence spectral data

```
class lumispypy.signals.el_spectrum.ELpectrum(*args, **kwargs)
```

Bases: `LumiSpectrum`

General 1D electroluminescence signal class

```
class lumispypy.signals.el_spectrum.LazyELpectrum(*args, **kwargs)
```

Bases: `LazySignal`, `ELpectrum`

General lazy 1D electroluminescence signal class

## lumispypy.signals.luminescence\_spectrum module

Signal class for Luminescence spectral data (1D).

```
class lumispypy.signals.luminescence_spectrum.LazyLumiSpectrum(*args, **kwargs)
```

Bases: `LazySignal`, `LumiSpectrum`

General lazy 1D luminescence signal class.

```
class lumispypy.signals.luminescence_spectrum.LumiSpectrum(*args, **kwargs)
```

Bases: `Signal1D`, `CommonLumi`

General 1D luminescence signal class.

```
def _convert_data(newaxis, factor, inplace, jacobian, data2, var2)
```

Utility function to perform the data and variance conversion for signal unit transformations.

```
def _reset_variance_linear_model()
```

Resets the variance linear model parameters to their default values, as they are not applicable any longer after a Jacobian transformation.

```
def centroid(signal_range=None, **kwargs)
```

Finds the centroid (center of mass) of a peak in the spectrum from the wavelength (or pixel number) and the intensity at each pixel value. It basically represents a "weighted average" of the peak.

## Notes

This function only works for a single peak. If you have multiple peaks, slice the signal beforehand or use the `signal_range` parameter.

TODO: Implement this function for multiple peaks (with the `npeaks` parameter) by finding the top 2 peaks from mean spectrum and then returning a signal with 2 com.

### Parameters

- **signal\_range** (*tuple of ints or floats, optional*) – A tuple representing the indices of the signal axis (start index, end index) where the peak is located. If the tuple contains int, it slices on index. If the tuple contains float, it slices on signal units (default HyperSpy `s.inav[:]` functionality).
- **kwargs** (*dictionary*) – For the `scipy.interpolate.interp1d` function.

### Returns

**signal** – Signal object containing the center of mass for every pixel. Depending on the dimensionality the type is `Signal2D` or a `BaseSignal` (for single spectrum).

### Return type

`Signal2D`, `BaseSignal`

**px\_to\_nm\_grating\_solver**(*gamma\_deg, deviation\_angle\_deg, focal\_length\_mm, ccd\_width\_mm, grating\_central\_wavelength\_nm, grating\_density\_gr\_mm, inplace=False*)

Converts signal axis of 1D signal (in pixels) to wavelength, solving the grating equation. See `lumispay.axes.solve_grating_equation` for more details.

### Parameters

- **gamma\_deg** (*float*) – Inclination angle between the focal plane and the centre of the grating (found experimentally from calibration). In degree.
- **deviation\_angle\_deg** (*float*) – Also known as included angle. It is defined as the difference between angle of diffraction ( $\beta$ ) and angle of incidence ( $\alpha$ ). Given by manufacturer specsheet. In degree.
- **focal\_length\_mm** (*float*) – Given by manufacturer specsheet. In mm.
- **ccd\_width\_mm** (*float*) – The width of the CDD. Given by manufacturer specsheet. In mm.
- **grating\_central\_wavelength\_nm** (*float*) – Wavelength at the centre of the grating, where exit slit is placed. In nm.
- **grating\_density\_gr\_mm** (*int*) – Grating density in gratings per mm.
- **inplace** (*bool*) – If False, it returns a new object with the transformation. If True, the original object is transformed, returning no object.

### Returns

**signal** – A signal with calibrated wavelength units.

### Return type

`LumiSpectrum`

## Examples

```
>>> s = LumiSpectrum(np.ones(20),)
>>> s.px_to_nm_grating_solver(*params, inplace=True)
>>> s.axes_manager.signal_axes[0].units == 'nm'
```

**remove\_background\_from\_file**(*background=None, inplace=False, \*\*kwargs*)

Subtract the background to the signal in all navigation axes. If no background file is passed as argument, the *remove\_background()* from HyperSpy is called with the GUI.

### Parameters

- **background** (*array shape (2, n) or SignalID*) – An array containing the background x-axis and the intensity values *[[xs],[ys]]* or a *SignalID* object. If the x-axis values do not match the *signal\_axes*, then interpolation is done before subtraction. If only the intensity values are provided, *[ys]*, the functions assumes no interpolation needed.
- **inplace** (*boolean*) – If *False*, it returns a new object with the transformation. If *True*, the original object is transformed, returning no object.

### Returns

**signal** – A background subtracted signal.

### Return type

*LumiSpectrum*

## Notes

This function does not work with non-uniform axes.

**savetxt**(*filename, fmt='%0.5f', delimiter='\t', axes=True, transpose=False, \*\*kwargs*)

Writes luminescence spectrum object to simple text file.

Writes single spectra to a two-column data file with signal axis as X and data as Y. Writes linescan data to file with signal axis as first row and navigation axis as first column (flipped if *transpose=True*).

### Parameters

- **filename** (*string*) –
- **fmt** (*str or sequence of strs, optional*) – A single or sequence of format strings. Default is *'%0.5f'*.
- **delimiter** (*str, optional*) – String or character separating columns. Default is *'\t'*.
- **axes** (*bool, optional*) – If *True* (default), include axes in saved file. If *False*, save the data array only.
- **transpose** (*bool, optional*) – If *True*, transpose data array and exchange axes. Default is *false*. Ignored for single spectra.
- **\*\*kwargs** – Takes any additional arguments of *numpy.loadtxt*, e.g. *newline header, footer, comments, or encoding*.

### See also:

[numpy.savetxt](#)

## Examples

```

>>> import lumispy as lum
>>> import numpy as np
...
>>> # Spectrum:
>>> S = lum.signals.LumiSpectrum(np.arange(5))
>>> S.savetxt('spectrum.txt')
0.00000    0.00000
1.00000    1.00000
2.00000    2.00000
3.00000    3.00000
4.00000    4.00000
...
>>> # Linescan:
>>> L = lum.signals.LumiSpectrum(np.arange(25).reshape((5,5)))
>>> L.savetxt('linescan.txt')
0.00000    0.00000  1.00000  2.00000  3.00000  4.00000
0.00000    0.00000  5.00000  10.00000      15.00000      20.00000
1.00000    1.00000  6.00000  11.00000      16.00000      21.00000
2.00000    2.00000  7.00000  12.00000      17.00000      22.00000
3.00000    3.00000  8.00000  13.00000      18.00000      23.00000
4.00000    4.00000  9.00000  14.00000      19.00000      24.00000

```

`to_array(axes=True, transpose=False)`

Returns luminescence spectrum object as numpy array (optionally including the axes).

Returns single spectra as two-column array. Returns linescan data as array with signal axis as first row and navigation axis as first column (flipped if `transpose=True`).

### Parameters

- **axes** (*bool*, *optional*) – If True (default), include axes in array. If False, return the data array only.
- **transpose** (*bool*, *optional*) – If True, transpose data array and exchange axes. Default is false. Ignored for single spectra.
- **\*\*kwargs** – Takes any additional arguments of `numpy.loadtxt`, e.g. *newline header*, *footer*, *comments*, or *encoding*.

### Notes

The output of this function can be used to convert a signal object to a pandas dataframe, e.g. using `df = pd.DataFrame(S.to_array())`.

## Examples

```

>>> import lumispy as lum
>>> import numpy as np
...
>>> # Spectrum:
>>> S = lum.signals.LumiSpectrum(np.arange(5))
>>> S.to_array()
array([[0., 0.],
       [1., 1.],
       [2., 2.],
       [3., 3.],
       [4., 4.]])
...
>>> # Linescan:
>>> L = lum.signals.LumiSpectrum(np.arange(25).reshape((5,5)))
>>> L.to_array()
array([[ 0.,  0.,  1.,  2.,  3.,  4.],
       [ 0.,  0.,  1.,  2.,  3.,  4.],
       [ 1.,  5.,  6.,  7.,  8.,  9.],
       [ 2., 10., 11., 12., 13., 14.],
       [ 3., 15., 16., 17., 18., 19.],
       [ 4., 20., 21., 22., 23., 24.]])

```

### `to_eV(inplace=True, jacobian=True)`

Converts signal axis of 1D signal to non-linear energy axis (eV) using wavelength dependent refractive index of air. Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/meV by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, J. Phys. Chem. Lett. 4, 3316 (2013), doi:10.1021/jz401508t, which ensures that integrated signals are correct also in the energy domain. If the variance of the signal is known, i.e. `metadata.Signal.Noise_properties.variance` is a signal representing the variance, a squared renormalization of the variance is performed. Note that if the variance is a number (not a signal instance), it is converted to a signal if the Jacobian transformation is performed

#### Parameters

- **inplace** (*boolean*) – If *False*, a new signal object is created and returned. Otherwise (default) the operation is performed on the existing signal object.
- **jacobian** (*boolean*) – The default is to do the Jacobian transformation (recommended at least for luminescence signals), but the transformation can be suppressed by setting this option to *False*.

## Examples

```

>>> import numpy as np
>>> from lumispy import LumiSpectrum
>>> S1 = LumiSpectrum(np.ones(20), DataAxis(axis = np.arange(200,400,10)), )
>>> S1.to_eV()

```

### `to_invcm(inplace=True, jacobian=True)`

Converts signal axis of 1D signal to non-linear wavenumber axis ( $\text{cm}^{-1}$ ). Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/ $\text{cm}^{-1}$  by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, J. Phys. Chem. Lett. 4, 3316 (2013), doi:10.1021/jz401508t, which ensures that integrated signals are correct also in the wavenumber domain. If the variance of the signal is known, i.e. `metadata.Signal.Noise_properties.variance` is a signal representing the variance, a squared renormalization of the variance is performed. Note that if the variance is a number (not a signal instance), it is converted to a signal if the Jacobian transformation is performed

### Parameters

- **inplace** (*boolean*) – If *False*, a new signal object is created and returned. Otherwise (default) the operation is performed on the existing signal object.
- **jacobian** (*boolean*) – The default is to do the Jacobian transformation (recommended at least for luminescence signals), but the transformation can be suppressed by setting this option to *False*.

### Examples

```
>>> import numpy as np
>>> from lumispy import LumiSpectrum
>>> S1 = LumiSpectrum(np.ones(20), DataAxis(axis = np.arange(200,400,10)), )
>>> S1.to_invcm()
```

**to\_invcm\_relative**(*laser=None, inplace=True, jacobian=False*)

Converts signal axis of 1D signal to non-linear wavenumber axis ( $\text{cm}^{-1}$ ) relative to the exciting laser wavelength (Raman/Stokes shift). Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/ $\text{cm}^{-1}$  by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, J. Phys. Chem. Lett. 4, 3316 (2013), doi:10.1021/jz401508t, which ensures that integrated signals are correct also in the wavenumber domain. If the variance of the signal is known, i.e. `metadata.Signal.Noise_properties.variance` is a signal representing the variance, a squared renormalization of the variance is performed. Note that if the variance is a number (not a signal instance), it is converted to a signal if the Jacobian transformation is performed

### Parameters

- **inplace** (*boolean*) – If *False*, a new signal object is created and returned. Otherwise (default) the operation is performed on the existing signal object.
- **laser** (*float or None*) – Laser wavelength in the same units as the signal axis. If *None* (default), checks if it is stored in `metadata.Acquisition_instrument.Laser.wavelength`.
- **jacobian** (*boolean*) – The default is not to do the Jacobian transformation for Raman shifts, but the transformation can be activated by setting this option to *True*.

### Examples

```
>>> import numpy as np
>>> from lumispy import LumiSpectrum
>>> S1 = LumiSpectrum(np.ones(20), DataAxis(axis = np.arange(200,400,10)), )
>>> S1.to_invcm(laser=325)
```

**to\_raman\_shift**(*laser=None, inplace=True, jacobian=False*)

Converts signal axis of 1D signal to non-linear wavenumber axis ( $\text{cm}^{-1}$ ) relative to the exciting laser wavelength (Raman/Stokes shift). Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/ $\text{cm}^{-1}$  by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, J. Phys. Chem. Lett. 4, 3316 (2013), doi:10.1021/jz401508t, which ensures that integrated signals are correct also in the wavenumber domain. If the variance of the signal is known, i.e. *metadata.Signal.Noise\_properties.variance* is a signal representing the variance, a squared renormalization of the variance is performed. Note that if the variance is a number (not a signal instance), it is converted to a signal if the Jacobian transformation is performed

#### Parameters

- **inplace** (*boolean*) – If *False*, a new signal object is created and returned. Otherwise (default) the operation is performed on the existing signal object.
- **laser** (*float or None*) – Laser wavelength in the same units as the signal axis. If *None* (default), checks if it is stored in *metadata.Acquisition\_instrument.Laser.wavelength*.
- **jacobian** (*boolean*) – The default is not to do the Jacobian transformation for Raman shifts, but the transformation can be activated by setting this option to *True*.

#### Examples

```
>>> import numpy as np
>>> from lumispy import LumiSpectrum
>>> S1 = LumiSpectrum(np.ones(20), DataAxis(axis = np.arange(200,400,10)), )
>>> S1.to_invcm(laser=325)
```

## lumispy.signals.luminescence\_transient module

### Signal class for luminescence transient data (1D)

**class** lumispy.signals.luminescence\_transient.LazyLumiTransient(\*args, \*\*kwargs)

Bases: *LazySignal, LumiTransient*

General lazy 1D luminescence signal class (transient/time resolved)

**class** lumispy.signals.luminescence\_transient.LumiTransient(\*args, \*\*kwargs)

Bases: *Signal1D, CommonTransient*

General 1D luminescence signal class (transient/time resolved)

## lumispy.signals.luminescence\_transientspec module

### Signal class for luminescence transient data (2D)

**class** lumispy.signals.luminescence\_transientspec.LazyLumiTransientSpectrum(\*args, \*\*kwargs)

Bases: *LazySignal, LumiTransientSpectrum*

General lazy 2D luminescence signal class (transient/time resolved)

**class** lumispy.signals.luminescence\_transient\_spec.**LumiTransientSpectrum**(\*args, \*\*kwargs)  
 Bases: *Signal2D, CommonLumi, CommonTransient*  
 General 2D luminescence signal class (transient/time resolved)

## lumispy.signals.pl\_spectrum module

### Signal class for Photoluminescence spectral data

**class** lumispy.signals.pl\_spectrum.**LazyPLSpectrum**(\*args, \*\*kwargs)  
 Bases: *LazySignal, PLSpectrum*  
 General lazy 1D photoluminescence signal class

**class** lumispy.signals.pl\_spectrum.**PLSpectrum**(\*args, \*\*kwargs)  
 Bases: *LumiSpectrum*  
 General 1D photoluminescence signal class

## Module contents

### lumispy.utils package

#### Submodules

#### lumispy.utils.axes module

**lumispy.utils.axes.n\_air**(x)

Refractive index of air as a function of WL in nm. This analytical function is correct for the range 185-1700 nm. According to *E.R. Peck and K. Reeder. Dispersion of air, J. Opt. Soc. Am. 62, 958-962 (1972)*.

**lumispy.utils.axes.axis2eV**(ax0)

Converts given signal axis to energy scale (eV) using wavelength dependent refractive index of air. Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

**lumispy.utils.axes.axis2invcm**(ax0)

Converts given signal axis to wavenumber scale ( $\text{cm}^{-1}$ ). Assumes wavelength in units of nm unless the axis units are specifically set to  $\mu\text{m}$ .

**lumispy.utils.axes.data2eV**(data, factor, evaxis, ax0)

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/meV by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, *J. Phys. Chem. Lett.* 4, 3316 (2013). Ensures that integrated signals are still correct.

**lumispy.utils.axes.data2invcm**(data, factor, invcmaxis, ax0=None)

The intensity is converted from counts/nm (counts/ $\mu\text{m}$ ) to counts/ $\text{cm}^{-1}$  by doing a Jacobian transformation, see e.g. Mooney and Kambhampati, *J. Phys. Chem. Lett.* 4, 3316 (2013). Ensures that integrated signals are still correct.

**lumispy.utils.axes.eV2nm**(x)

Converts energy (eV) to wavelength (nm) using wavelength-dependent refractive index of air.

`lumispy.utils.axes.invcn2nm(x)`

Converts wavenumber ( $\text{cm}^{-1}$ ) to wavelength (nm).

`lumispy.utils.axes.join_spectra(S, r=50, scale=True, average=False, kind='slinear')`

Takes list of `Signal1D` objects and returns a single object with all spectra joined. Joins spectra at the center of the overlapping range. Scales spectra by a factor determined as average over the range *center* +/- *r* pixels. Works both for uniform and non-uniform axes (`FunctionalDataAxis` is converted into a non-uniform `DataAxis`).

#### Parameters

- **S** (*list of Signal1D objects (with overlapping signal axes)*) –
- **r** (*int, optional*) – Number of pixels left/right of center (default 50) defining the range over which to determine the scaling factor, has to be less than half of the overlapping pixels. Change the size of *r* or use *average=True* if the function induces a step in the intensity.
- **scale** (*boolean, optional*) – If *True* (default), the later spectra in the list are scaled by a factor determined over *center* +/- *r* pixels. If *False*, spectra are joined without scaling, which will likely induce a step unless *average=True*.
- **average** (*boolean, optional*) – If *True*, the contribution of the two signals is continuously graded within the range defined by *r* instead of joining at the center of the range (default).
- **kind** (*str, optional*) – Interpolation method (default 'slinear') to use when joining signals with a uniform signal axes. See `scipy.interpolate.interp1d` for options.

#### Returns

- A new `Signal1D` object containing the joined spectra (properties are copied
- from first spectrum).

#### Examples

```
>>> s1 = hs.signals.Signal1D(np.ones(32))
>>> s2 = hs.signals.Signal1D(np.ones(32)*2)
>>> s2.axes_manager.signal_axes[0].offset = 25
>>> lum.join_spectra([s1,s2],r=2)
<Signal1D, title: , dimensions: (|57)>
```

`lumispy.utils.axes.nm2eV(x)`

Converts wavelength (nm) to energy (eV) using wavelength-dependent refractive index of air.

`lumispy.utils.axes.nm2invcn(x)`

Converts wavelength (nm) to wavenumber ( $\text{cm}^{-1}$ ).

`lumispy.utils.axes.solve_grating_equation(axis, gamma_deg, deviation_angle_deg, focal_length_mm, ccd_width_mm, grating_central_wavelength_nm, grating_density_gr_mm)`

Solves the grating equation. See [horiba.com/uk/scientific/products/optics-tutorial/wavelength-pixel-position](http://horiba.com/uk/scientific/products/optics-tutorial/wavelength-pixel-position) for equations.

#### Parameters

- **axis** (*hyperspy.axis*) – Axis in pixel units (no units) to convert to wavelength.
- **gamma\_deg** (*float*) – Inclination angle between the focal plane and the centre of the grating (found experimentally from calibration). In degree.

- **deviation\_angle\_deg** (*float*) – Also known as included angle. It is defined as the difference between angle of diffraction ( $\beta$ ) and angle of incidence ( $\alpha$ ). Given by manufacturer specsheet. In degree.
- **focal\_length\_mm** (*float*) – Given by manufacturer specsheet. In mm.
- **ccd\_width\_mm** (*float*) – The width of the CDD. Given by manufacturer specsheet. In mm.
- **grating\_central\_wavelength\_nm** (*float*) – Wavelength at the centre of the grating, where exit slit is placed. In nm.
- **grating\_density\_gr\_mm** (*int*) – Grating density in gratings per mm.

**Returns**

**axis** – HyperSpy axis object.

**Return type**

hyperspy.axis

`lumispay.utils.axes.var2eV`(*variance, factor, evaxis, ax0*)

The variance is converted doing a squared Jacobian renormalization to match with the transformation of the data.

`lumispay.utils.axes.var2invcm`(*variance, factor, invcmaxis, ax0=None*)

The variance is converted doing a squared Jacobian renormalization to match with the transformation of the data.

**lumispay.utils.io module**

`lumispay.utils.io.savetxt`(*S, filename, fmt='%0.5f', delimiter='\n', axes=True, transpose=False, \*\*kwargs*)

Writes signal object to simple text file.

Writes single spectra to a two-column data file with signal axis as X and data as Y. Writes linescan data to file with signal axis as first row and navigation axis as first column (flipped if *transpose=True*). Writes image to file with the navigation axes as first column and first row. Writes 2D data (e.g. map of a fit parameter value) to file with the signal axes as first column and first row.

**Parameters**

- **filename** (*string*) –
- **fmt** (*str or sequence of strs, optional*) – A single or sequence of format strings. Default is `'%0.5f'`.
- **delimiter** (*str, optional*) – String or character separating columns. Default is `'\n'`.
- **axes** (*bool, optional*) – If True (default), include axes in saved file. If False, save the data array only.
- **transpose** (*bool, optional*) – If True, transpose data array and exchange axes. Default is false. Ignored for single spectra.
- **\*\*kwargs** – Takes any additional arguments of `numpy.loadtxt`, e.g. *newline header, footer, comments, or encoding*.

**See also:**

`numpy.savetxt`

## Examples

```

>>> import lumispy as lum
>>> import numpy as np
...
>>> # Spectrum:
>>> S = lum.signals.LumiSpectrum(np.arange(5))
>>> lum.savetxt(S, 'spectrum.txt')
0.00000    0.00000
1.00000    1.00000
2.00000    2.00000
3.00000    3.00000
4.00000    4.00000
...
>>> # Linescan:
>>> L = lum.signals.LumiSpectrum(np.arange(25).reshape((5,5)))
>>> lum.savetxt(L, 'linescan.txt')
0.00000    0.00000  1.00000  2.00000  3.00000  4.00000
0.00000    0.00000  5.00000  10.00000      15.00000      20.00000
1.00000    1.00000  6.00000  11.00000      16.00000      21.00000
2.00000    2.00000  7.00000  12.00000      17.00000      22.00000
3.00000    3.00000  8.00000  13.00000      18.00000      23.00000
4.00000    4.00000  9.00000  14.00000      19.00000      24.00000

```

`lumispy.utils.io.to_array(S, axes=True, transpose=False)`

Returns signal object as numpy array (optionally including the axes).

Returns single spectra as two-column array. Returns linescan data as array with signal axis as first row and navigation axis as first column (flipped if `transpose=True`). Returns image as array with the navigation axes as first column and first row. Returns 2D data (e.g. map of a fit parameter value) as array with the signal axes as first column and first row.

### Parameters

- **axes** (*bool*, *optional*) – If True (default), include axes in array. If False, return the data array only.
- **transpose** (*bool*, *optional*) – If True, transpose data array and exchange axes. Default is false. Ignored for single spectra.
- **\*\*kwargs** – Takes any additional arguments of `numpy.loadtxt`, e.g. *newline header*, *footer*, *comments*, or *encoding*.

### Notes

The output of this function can be used to convert a signal object to a pandas dataframe, e.g. using `df = pd.DataFrame(lum.to_array(S))`.

## Examples

```

>>> import lumispy as lum
>>> import numpy as np
...
>>> # Spectrum:
>>> S = lum.signals.LumiSpectrum(np.arange(5))
>>> lum.to_array(S)
array([[0., 0.],
       [1., 1.],
       [2., 2.],
       [3., 3.],
       [4., 4.]])
...
>>> # Linescan:
>>> L = lum.signals.LumiSpectrum(np.arange(25).reshape((5,5)))
>>> lum.to_array(L)
array([[ 0.,  0.,  1.,  2.,  3.,  4.],
       [ 0.,  0.,  1.,  2.,  3.,  4.],
       [ 1.,  5.,  6.,  7.,  8.,  9.],
       [ 2., 10., 11., 12., 13., 14.],
       [ 3., 15., 16., 17., 18., 19.],
       [ 4., 20., 21., 22., 23., 24.]])

```

## lumispy.utils.signals module

`lumispy.utils.signals.com(spectrum_intensities, signal_axis, **kwargs)`

Finds the centroid (center of mass) of a peak in the spectrum based from the intensity at each pixel value and its respective signal axis.

### Parameters

- **spectrum\_intensities** (*array*) – An array with the intensities of the spectrum.
- **signal\_axis** (*hyperspy.axes.BaseDataAxis subclass*) – A HyperSpy signal axis class containing an array with the wavelength/ energy for each intensity/signal value.
- **kwargs** (*dictionary*) – For the `scipy.interpolate.interp1d` function.

### Returns

**center\_of\_mass** – The centroid of the spectrum.

### Return type

`float`

## Examples

```
# Assume we have a spectrum with wavelengths and intensities >>> wavelengths = [200, 300, 400, 500, 600, 700] >>> intensities = [1, 2, 3, 2, 1, 0] >>> from hyperspy.axes import DataAxis >>> signal_axis = DataAxis(axis=wavelengths)
```

```
>>> center_of_mass = com(intensities, signal_axis)
>>> print(center_of_mass) # Outputs: [400.0]
```

## Module contents

## Module contents

# 1.10 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

## 1.10.1 2023-03-15 - version 0.2.2

### Changed

- Use [GitHub code scanning \(CodeQL\)](#) for integrity check as it replaces [LGTm](#)
- Added a centroid/center of mass functionality to analyse peak position of a spectrum (both in *utils`* and in *LumiSpectrum`*)
- Add documentation of signal tools

### Maintenance

- Replace `sphinx.ext.imgmath` by `sphinx.ext.mathjax` to fix the math rendering in the *ReadTheDocs* build
- fix external references in the documentation

## 1.10.2 2022-11-02 - version 0.2.1

### Added

- Improved documentation
- Use [lgtm.com](#) to check code integrity

## Changed

- Fix conversion to Raman shift (relative wavenumber) and make `jacobian=False` default; fix `inplace=False` for axis conversions
- Fix `to_eV` and `to_invcm`, as slicing with `.isig[]` was failing on converted signals
- `s.remove_negative` now defaults to `inplace=False` (previously True)

## Maintenance

- Use `softprops/action-gh-release` action instead of deprecated `create-release`, pin action to a commit SHA

### 1.10.3 2022-04-29 - version 0.2.0

#### Added

- Set up read the docs documentation
- Added metadata convention
- Add proper handling of variance on Jacobian transformation during axis conversion (eV, invcm)

#### Changed

- Account for the general availability of non-uniform axes with the HyperSpy v1.7 release
- Make `LumiTransient` 1D and add 2D `LumiTransientSpectrum` class
- Add python 3.10 build, remove python 3.6
- Fix error in background dimensions that allows compatibility for updated `map` in HyperSpy (failing integration tests)
- Fix for links in PyPi
- Deprecate `exposure` argument of `s.scale_by_exposure` in favor of `integration_time` in line with meta-data convention
- Add deprecation warning to `remove_background_from_file`

### 1.10.4 2021-11-23 - version 0.1.3

#### Changed

- Mentions of the now deleted `non_uniform_axes` branch in HyperSpy updated to `RELEASE_next_minor`
- Change 'master' to 'main' branch
- Updated/corrected badges and other things in README.md and other documentation files

### 1.10.5 2021-08-22 - version 0.1.2

#### Added

- This project now keeps a Changelog
- Added signal-hierarchy for time-resolved luminescence
- Added GitHub action for release
- Created logo

#### Changed

- Consistent black-formatting
- fixed join\_spectra
- fixed tests

### 1.10.6 2021-03-26 - version 0.1.0

#### Added

- The first release, basic functionality implemented

## 1.11 Citing LumiSpy

LumiSpy is maintained by an [active community of developers](#).

If LumiSpy has been significant to a project that leads to an academic publication, please acknowledge that fact by citing it. The DOI in the badge below is the [Concept DOI](#) – it can be used to cite the project without referring to a specific version. If you are citing LumiSpy because you have used it to process data, please use the DOI of the specific version that you have employed. You can find it by clicking on the DOI badge below:

## 1.12 Contributing

LumiSpy is meant to be a community maintained project. We welcome contributions in the form of bug reports, documentation, code, feature requests, and more. In the following we refer to some resources to help you make useful contributions.

### 1.12.1 Issues

The [issue tracker](#) can be used to report bugs or propose new features. When reporting a bug, the following is useful:

- give a minimal example demonstrating the bug,
- copy and paste the error traceback.

### 1.12.2 Pull Requests

If you want to contribute to the LumiSpy source code, you can send us a [pull request](#). Small bug fixes are corrections to the user guide are typically a good starting point. But don't hesitate also for significant code contributions - if needed, we'll help you to get the code ready to common standards.

Please refer to the [HyperSpy developer guide](#) in order to get started and for detailed contributing guidelines.

The [kikuchipy contributors guide](#), another HyperSpy extension, also is a valuable resource that can get you started and provides useful guidelines.

### Reviewing

As quality assurance, to improve the code, and to ensure a generalized functionality, pull requests need to be thoroughly reviewed by at least one other member of the development team before being merged.

### 1.12.3 Documentation

The LumiSpy documentation consists of three elements:

- Docstrings following the [numpy standard](#) that document the functionality of individual methods on [GitHub](#).
- The [documentation](#) written using [Sphinx](#) and hosted on [Read the Docs](#). The source is part of the [GitHub repository](#).
- A set of curated Jupyter notebooks in the [LumiSpy demos repository](#) on [GitHub](#) that provide tutorials and example workflows.

Improving documentation is always welcome and a good way of starting out to learn the [GitHub](#) functionality. You can contribute through pull requests to the respective repositories.

### 1.12.4 Code style

LumiSpy follows [Style Guide for Python Code](#) with [The Black Code style](#).

For [docstrings](#), we follow the [numpydoc](#) standard.

Package imports should be structured into three blocks with blank lines between them:

- standard libraries (like `os` and `typing`),
- third party packages (like `numpy` and `hyperspy`),
- and finally `lumi spy` imports.

### 1.12.5 Writing tests

All functionality in LumiSpy is tested via the [pytest](#) framework. The tests reside in the `test` directory. Tests are short methods that call functions in LumiSpy and compare resulting output values with known answers. Please refer to the [HyperSpy development guide](#) for further information on tests.

### 1.12.6 Releasing a new version

LumiSpy versioning follows [semantic versioning](#) and the version number is therefore a three-part number: MAJOR.MINOR.PATCH. Each number will change depending on the type of changes according to the following:

- MAJOR increases when making incompatible API changes,
- MINOR increases when adding functionality in a backwards compatible manner, and
- PATCH increases when making backwards compatible bug fixes.

The process to release a new version that is pushed to [PyPI](#) and [Conda-Forge](#) is documented in the [Releasing guide](#).

## 1.13 License

LumiSpy is free software: you can redistribute it and/or modify it under the terms of the [GNU General Public License \(GPL\)](#) as published by the Free Software Foundation, either version 3 of the license, or (at your option) any later version.

LumiSpy is distributed in the hope that it will be useful, but **without any warranty**; without even the implied warranty of **merchantability** or **fitness for a particular purpose**. See the [GNU General Public License](#) for more details.

## BIBLIOGRAPHY

- [Mooney] J. Mooney and P. Kambhampati, *The Journal of Physical Chemistry Letters* **4**, 3316 (2013).  
[doi:10.1021/jz401508t](https://doi.org/10.1021/jz401508t)
- [Peck] E.R. Peck and K. Reeder, *J. Opt. Soc. Am.* **62**, 958 (1972). [doi:10.1364/JOSA.62.000958](https://doi.org/10.1364/JOSA.62.000958)
- [Pfueller] C. Pfüller, Dissertation, HU Berlin, p. 28 (2011). [doi:10.18452/16360](https://doi.org/10.18452/16360)
- [Tappy] N. Tappy, A. Fontcuberta i Morral and C. Monachon, *Rev. Sci. Instrum.* **93**, 053702 (2022).  
[doi:10.1063/5.0080486](https://doi.org/10.1063/5.0080486)
- [Wang] Y. Wang and P. D. Townsend, *J. Luminesc.* **142**, 202 (2013). [doi:10.1016/j.jlumin.2013.03.052](https://doi.org/10.1016/j.jlumin.2013.03.052)



## PYTHON MODULE INDEX

|

`lumispy`, 34  
`lumispy.signals`, 29  
`lumispy.signals.cl_spectrum`, 18  
`lumispy.signals.common_luminescence`, 20  
`lumispy.signals.common_transient`, 22  
`lumispy.signals.el_spectrum`, 22  
`lumispy.signals.luminescence_spectrum`, 22  
`lumispy.signals.luminescence_transient`, 28  
`lumispy.signals.luminescence_transientspec`,  
28  
`lumispy.signals.pl_spectrum`, 29  
`lumispy.utils`, 34  
`lumispy.utils.axes`, 29  
`lumispy.utils.io`, 31  
`lumispy.utils.signals`, 33



Symbols

- `_convert_data()` (*lumisp.py.signals.luminescence\_spectrum.LumiSpectrum* method), 22
  - `_make_signal_mask()` (*lumisp.py.signals.cl\_spectrum.CLpectrum* method), 19
  - `_n_air()` (*in module lumisp.py.utils.axes*), 29
  - `_reset_variance_linear_model()` (*lumisp.py.signals.luminescence\_spectrum.LumiSpectrum* method), 22
- A**
- `axis2eV()` (*in module lumisp.py.utils.axes*), 29
  - `axis2invcm()` (*in module lumisp.py.utils.axes*), 29
- C**
- `centroid()` (*lumisp.py.signals.luminescence\_spectrum.LumiSpectrum* method), 22
  - `CLSEMSpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 18
  - `CLpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 19
  - `CLSTEMSpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 19
  - `com()` (*in module lumisp.py.utils.signals*), 33
  - `CommonLumi` (*class in lumisp.py.signals.common\_luminescence*), 20
  - `CommonTransient` (*class in lumisp.py.signals.common\_transient*), 22
  - `correct_grating_shift()` (*lumisp.py.signals.cl\_spectrum.CLSEMSpectrum* method), 18
  - `crop_edges()` (*lumisp.py.signals.common\_luminescence.CommonLumi* method), 20
- D**
- `data2eV()` (*in module lumisp.py.utils.axes*), 29
  - `data2invcm()` (*in module lumisp.py.utils.axes*), 29
- E**
- `ELSpectrum` (*class in lumisp.py.signals.el\_spectrum*), 22
  - `eV2nm()` (*in module lumisp.py.utils.axes*), 29
- I**
- `invcm2nm()` (*in module lumisp.py.utils.axes*), 29
- J**
- `join_spectra()` (*in module lumisp.py.utils.axes*), 30
- L**
- `LazyCLSEMSpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 20
  - `LazyCLpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 20
  - `LazyCLSTEMSpectrum` (*class in lumisp.py.signals.cl\_spectrum*), 20
  - `LazyELSpectrum` (*class in lumisp.py.signals.el\_spectrum*), 22
  - `LazyLumiSpectrum` (*class in lumisp.py.signals.luminescence\_spectrum*), 22
  - `LazyLumiTransient` (*class in lumisp.py.signals.luminescence\_transient*), 28
  - `LazyLumiTransientSpectrum` (*class in lumisp.py.signals.luminescence\_transientspec*), 28
  - `LazyPLSpectrum` (*class in lumisp.py.signals.pl\_spectrum*), 29
  - `LumiSpectrum` (*class in lumisp.py.signals.luminescence\_spectrum*), 22
- lumisp**
- module, 34
- lumisp.py**
- `signals` module, 29
  - `signals.cl_spectrum` module, 18
  - `signals.common_luminescence` module, 20
  - `signals.common_transient` module, 22
  - `signals.el_spectrum` module, 22
  - `signals.luminescence_spectrum` module, 22

lumispy.signals.luminescence\_transient  
     module, 28  
 lumispy.signals.luminescence\_transientspec  
     module, 28  
 lumispy.signals.pl\_spectrum  
     module, 29  
 lumispy.utils  
     module, 34  
 lumispy.utils.axes  
     module, 29  
 lumispy.utils.io  
     module, 31  
 lumispy.utils.signals  
     module, 33  
 LumiTransient (class in lumispy.signals.luminescence\_transient), 28  
 LumiTransientSpectrum (class in lumispy.signals.luminescence\_transientspec), 28

## M

module  
     lumispy, 34  
     lumispy.signals, 29  
     lumispy.signals.cl\_spectrum, 18  
     lumispy.signals.common\_luminescence, 20  
     lumispy.signals.common\_transient, 22  
     lumispy.signals.el\_spectrum, 22  
     lumispy.signals.luminescence\_spectrum, 22  
     lumispy.signals.luminescence\_transient, 28  
     lumispy.signals.luminescence\_transientspec, 28  
     lumispy.signals.pl\_spectrum, 29  
     lumispy.utils, 34  
     lumispy.utils.axes, 29  
     lumispy.utils.io, 31  
     lumispy.utils.signals, 33

## N

nm2eV() (in module lumispy.utils.axes), 30  
 nm2invcm() (in module lumispy.utils.axes), 30  
 normalize() (lumispy.signals.common\_luminescence.CommonLumi method), 20

## P

PLSpectrum (class in lumispy.signals.pl\_spectrum), 29  
 px\_to\_nm\_grating\_solver() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 23

## R

remove\_background\_from\_file() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 24

remove\_negative() (lumispy.signals.common\_luminescence.CommonLumi method), 21  
 remove\_spikes() (lumispy.signals.cl\_spectrum.CLSpectrum method), 19

## S

savetxt() (in module lumispy.utils.io), 31  
 savetxt() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 24  
 scale\_by\_exposure() (lumispy.signals.common\_luminescence.CommonLumi method), 21  
 solve\_grating\_equation() (in module lumispy.utils.axes), 30

## T

to\_array() (in module lumispy.utils.io), 32  
 to\_array() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 25  
 to\_eV() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 26  
 to\_invcm() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 26  
 to\_invcm\_relative() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 27  
 to\_raman\_shift() (lumispy.signals.luminescence\_spectrum.LumiSpectrum method), 27

## V

var2eV() (in module lumispy.utils.axes), 31  
 var2invcm() (in module lumispy.utils.axes), 31