align_sped Documentation

Release 0.1

Emil Christiansen

February 08, 2017

CONTENTS

Contents:

CHAPTER

ONE

DOCUMENTATION OF *SPED_ALIGN.PY*

This documentation describes some tools implemented in Python to pre-process Scanning Precession Electron Diffraction (SPED) data. So far, only procedures for aligning stacks of diffraction patterns are implemented.

Please note that this module depends on the hyperspy.api package available at http://hyperspy.org/.

1.1 Disclaimer:

The code has not been significantly tested and its robustness cannot be guaranteed. When using this software with e.g. scientific data, be sure to backup your raw data first, in the case this software tampers with things it really should not I am not responsible for any eventual artefacts introduced by the code on scientific data, or the loss of eventual data and/or metadata.

1.2 Aligning a SPED stack

When acquiring a SPED data set, the diffraction pattern identified with each scan pixel may sometimes be displaced relative to other frames. Whether this is due to misalignments in the instrument, probe-sample interaction (such as thickness), or any other phenomena, it is sometimes neccessary to correct these relative shifts. This is especially true if one wants to use e.g. the ASTAR software *index*'+'*mapviewer* from NanoMEGAS, as the center-fitting algorithm in *index* is painfully slow and can easily loose the direct beam and wander off.

The function *align_stack_fast* attempts to fit a twodimensional gaussian to a subset of each frame, and to use the positions of the origin of this as a measure of the shift. The gaussian has to be a "true" 2D gaussian in the sense that it must have two independent "widths" as well as the possibility to rotate. This is because the direct beam sometimes "deforms" and becoms elliptical in a general direction. It is worth noting, that since the Stingray camera in the microscope setup saturates at 255, the (000) reflection are usually saturated and a more "general" fitting function would probably be a fermi-dirac function. The possibility for using this as a fit instead may be implemented in the future.

sped_align.**align_stack_fast**(*stack*, *limit=None*, *bounds=10*, *save=False*, *savedir=None*, *name=None*)

Find 000 position of each frame in stack and align the stack to these positions

Parameters

- **stack** (*hyperspy.api.signals.Signal2D*) – The stack containing the diffraction patterns to be aligned.
- **limit** (*int, float or str, optional*) – The total number of frames to consider before ending, useful for debugging (default is None which implies treating the whole stack).

- **bounds** (*int, float or str, optional*) – Defines the region where a 2D gaussian is fitted to the dataset (the default is 10). The region is defined as *frame[center[0]-bounds:center[0]+bounds+1, center[1]-bounds:center[1]+bounds+1*. The region should be chosen large enough to (preferably) only contain the 000 reflection.
- **save** (*{False, True}, optional*) – Whether or not to save the aligned stack and the gaussian fits as hdf5 files.
- **savedir** (*str, optional*) – The directory to store the results, if *save=True* (defaults to '' which implies storing results in the cwd).
- **name** (*str, optional*) – The name to give the aligned stack (defaults to *aligned_stack*).

Returns

- **Popt** (*numpy.ndarray*) – The fit results containing all the data to produce the fitted gaussian. The array has shape (N*M, 7) where N and M are the scan dimensions of the stack (total number of frames), and 7 is the number of parameters to describe the gaussian.
- **g** (*hyperspy.api.signals.Signal2D*) – A stack of the intensities of the fitted gaussians.
- **aligned_stack** (*hyperspy.api.signals.Signal2D*) – A stack containing the aligned diffraction patterns of the input stack.

See also:

gaussian_2d_fast() Compute 2D gaussian

fit_gaussian_2d_to_imagesubset_fast() Fit a 2D gaussian to a subset of image

Examples

Load a blockfile containing Scanning Precession Electron Diffraction data and align them.

```
>>> s = hs.load('data.blo')
>>> Popts, g, s_aligned = sa.align_stack_fast(s, limit=None, save=True)
```

1.3 Fitting a Gaussian

In order to find the (000) reflection in a diffraction pattern, a Gaussian is fitted to a subset of the pattern. In other words: the routine attempts to find the best possible gaussian that suits the intensity in a certain region of the image. So far, only a simple implementation has been done, which assumes that the direct beam should be fairly close to the center of the pattern, and that the pattern is 144x144 pixels. The region that the gaussian will be fitted to is then a square of a specified width (half-width really) centered on the image center. In future versions, this region may be able to move, possibly by a user specified amount, in order to keep up with the direct beam.

Also worth pointing out is that the *scipy.optimize.curve_fit()* procedure has a *bounds* parameter that can be used to limit the parameter space it tests. In the present implementation, these bounds has been defined as *([max_value/2, 0, 0, 0, 0, -45,0], [max_value * 2, width_x, width_y, width_x / 2, width_y / 2, 45, np.inf])*, where the amplitude bound, row_origin bound, column_origin bound, row_width bound, column_width bound, rotation and offset bounds are the first, second, third, fourth, fifth, sixth, and seventh elements of the lists respectively. The first list of the tuple defines the lower bounds, while the second list defines the upper bounds. Feel free to adjust these bounds when needed.

sped_align.**fit_gaussian_2d_to_imagesubset_fast**(*image*, *bounds=10*)

Fit a guassian to a subset of an image

Parameters

- **image** (*numpy.ndarray*) – The image to be fitted.
- **bounds** (*int or float, optional*) – The number of pixels to the right, left, top and bottom of the image center that should be considered when fitting the gaussian (defaults to 10).

Returns **popt** – Optimal values for the parameters so that the sum of the squared error of `f(xdata, *popt) - ydata` is minimized. See *scipy.optimize.curvefit()*

Return type numpy.ndarray

See also:

`scipy.optimize.curvefit()`

Examples

Provide an image of shape *(n, m)* (a superposition of two gaussians) and fit a gaussian to a square of widths 15 pixels around its center. Show the result

```
>>> n, m = 256, 512
>>> X, Y = numpy.mgrid[:n, :m]
>>> G = gaussian_2d_fast((X, Y), 1337, 144, 256, 10, 23, 14, 42)
>>> img = G.reshape(np.shape(X))
>>> A, C, origin, widths, rotation = 512, 71, (150, 300), (10, 23), 35
>>> img += gaussian_2d_fast((X, Y), 512, 150, 300, 10, 23, 35, 71)
>>> popt = fit_gaussian_2d_to_imagesubset_fast(G, bounds=15)
>>> fitted_gaussian = gaussian_2d_fast((X, Y), *popt)
>>> fitted_gaussian.reshape(np.shape(X))
>>> fig, ax = matplotlib.pyplot.subplots(1, 2)
>>> ax[0].imshow(G, interpolation='nearest')
>>> ax[1].imshow(fitted_gaussian, interpolation='nearest')
>>> ax[0].set_title('Original image')
>>> ax[1].set_title('Fitted gaussian')
>>> matplotlib.pyplot.show()
```

1.4 Generating a Gaussian

The actual function that produces the Gaussians has to satisfy certain conditions. First of all, the gaussian should be able to arbitrarily rotate, secondly, the gaussian should be able to be fitted, and thirdly, it should be as fast as possible. The first two points here are satisfied by the implemented function *gaussian_2d_fast()*, however, how "fast" it actually is has not been investigated as there are other bottlenecks in the procedure. The reason why the function takes in the coordinate matrices as two separate elements in a *tuple* is that this is a requirement from the *scipy.optimize.curve_fit* routine. Any function that should be fitted requires the free variable to be a "single" parameter. Additionally, it expects to get a 1D array out as a result (I think).

`sped_align.`**`gaussian_2d_fast`**(*xdata_tuple*, *amplitude*, *xo*, *yo*, *sigma_x*, *sigma_y*, *theta*, *offset*)

Compute a flattened gaussian at each coordinate

Parameters

- **xdata_tuple** (*2-tuple*) – Tuple containing coordinate matrices at which points the guassian should be computed. *X* (first element - rows) and *Y* (second element - columns) generated through e.g. *X, Y = numpy.mgrid[0:N, 0:M]*.
- **amplitude** (*float*) – The amplitude of the gaussian.
- **xo** (*float*) – The origin of the guassian in the *X* direction (row).

- **yo** (*float*) – The origin of the gaussian in the *Y* direction (column).
- **sigma_x** (*float*) – The "width" of the unrotated gaussian in the *X* direction (row).
- **sigma_y** (*float*) – The "width" of the unrotated gaussian in the *Y* direction (column).
- **theta** (*float*) – The rotation of the gaussian in degrees, measured clockwise.
- **offset** (*float*) – The baseline of the gaussian ($G(x \to \infty, y \to \infty)$).

Returns **g** – The values of the gaussian at the input coordinates as an unraveled array. See Examples for more information

Return type numpy.ndarray

Notes

The gaussian of amplitude A and baseline C is expressed as:

$$G(x, y, \theta) = C + A * \exp\left(x'\left(x, y, \theta\right)^2 + y\left(x, y, \theta\right)'^2\right),$$

where x' and y' are rotated according to conventional algebra:

$$\begin{bmatrix} x'\left(x, y, \theta\right)^2 \\ y'\left(x, y, \theta\right)^2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \bar{x}\left(x, y\right) \\ \bar{y}\left(x, y\right) \end{bmatrix}.$$

$\bar{x}\left(x, y\right)$ and $\bar{y}\left(x, y\right)$ are functions of the cartesian coordinates x and y through the substitution of:

$$\bar{x}\left(x, y\right) = \frac{\left(x - x_0\right)^2}{2\sigma_x} \quad \bar{y}\left(x, y\right) = \frac{\left(y - y_0\right)^2}{2\sigma_y},$$

where σ_x and σ_y are the widths of the resulting gaussian.

In the code, this matrix multiplication and the substitutions has been carried out "by hand". There might be some numerical issues considering the implementation.

Examples

Compute a rotated gaussian at the specified points:

```
>>> n, m = 256, 512
>>> X, Y = numpy.mgrid[:n, :m]
>>> A, C, origin, widths, rotation = 1337, 42, (144, 256), (10, 23), 14
>>> G = gaussian_2d_fast((X, Y), 1337, 144, 256, 10, 23, 14, 42)
>>> G = G.reshape(np.shape(X))
```

Make a supersposition of two gaussians and fit a 2D gaussian to this sample image:

```
>>> n, m = 256, 512
>>> X, Y = numpy.mgrid[:n, :m]
>>> A, C, origin, widths, rotation = 1337, 42, (144, 256), (10, 23), 14
>>> G = gaussian_2d_fast((X, Y), 1337, 144, 256, 10, 23, 14, 42)
>>> img = G.reshape(np.shape(X))
>>> A, C, origin, widths, rotation = 512, 71, (150, 300), (10, 23), 35
>>> img += gaussian_2d_fast((X, Y), 512, 150, 300, 10, 23, 35, 71)
>>> img = img.reshape(np.shape(X))
>>> popt, pcov = scipy.optimize.curve_fit(gaussian_2d_fast, (X, Y), G)
>>> fitted_gaussian = gaussian_2d_fast((X, Y), *popt)
```

```
>>> fitted_gaussian = fitted_gaussian.reshape(np.shape(X))
>>> fig, ax = matplotlib.pyplot.subplots(1, 2)
>>> ax[0].imshow(img, interpolation='nearest')
>>> ax[1].imshow(fitted_gaussian, interpolation='nearest')
>>> ax[0].set_title('Original image')
>>> ax[1].set_title('Fitted gaussian')
>>> matplotlib.pyplot.show()
```

1.5 Want to shift the gaussians?

Sometimes, when a SPED stack has been aligned, it might be interesting (or useful when debugging) to load the stack of gaussians that was generated and shift this one as well. In these cases, *shift_stack* does the job, but note that it is important that the same *Popts.npy* file that was generated when running *align_stack_fast* is loaded and used as input.

sped_align.**shift_stack**(*stack*, *popt*, *n_cols*, *n_rows*, *ndx*, *ndy*, *limit=None*, *save=False*, *savedir=None*, *name=None*)

Shift a stack

Shifts the input stack, using the second and third elements in popt, and makes a new stack of the shifted images.

Parameters

- **stack** (*hyperspy.api.signals.Signal2D*) – A stack of e.g. Precession Electron Diffraction patterns to be shifted. Assumed to have shape (Nx, Ny, nx, ny)
- **popt** (*numpy.ndarray*) – A numpy array of shape (Nx*Ny, 7), where the second and third elements are used as shifts in row and column directions, respectively
- **n_cols** (*int, float or str*) – Number of columns in the stack (equal to Ny)
- **n_rows** (*int, float or str*) – Number of rows in the stack (equal to Nx)
- **ndx** (*int, float or str*) – Number of vertical (?) pixels in the diffraction pattern
- **ndy** (*int, float or str*) – Number of horizontal (?) pixels in the diffraction pattern
- **limit** (*int, float, str or None, optional*) – Number of frames to shift before breaking, useful for debugging (the default is *None* indicating that all frames will be shifted)
- **save** (*{False, True}, optional*) – Whether or not the shifted stack should be saved
- **savedir** (*int, float or str, optional*) – Used to specify a directory to store the shifted stack if *save=True* (default is *''* indicating that the signal will be stored in the cwd)
- **name** (*int, float or str, optional*) – Used to specify the name of the signal if *save=True* (default is *'shifted_stack'*

Returns

- **shifted_signal** (*hyperspy.api.signals.Signal2D*) – The shifted stack as a hyperspy signal
- **times** (*list of floats*) – The time spent since the start of the for loop, sampled at every 1% of total frames.

Notes

Examples

Load a premade *popt* file and a signal to shift, then shift the signal

```
>>> s = hs.load('C:\Users\emilc\Desktop\SPED_align_laptop\2017_01_11_6060-20-1-C-4_001.blo')
>>> n_cols = s.axes_manager['x'].get('size')['size']
>>> n_rows = s.axes_manager['y'].get('size')['size']
>>> ndx = s.axes_manager['dx'].get('size')['size']
>>> ndy = s.axes_manager['dy'].get('size')['size']
>>> Popts = np.load(savedir+'Popts_1.npy')
>>> shifted_signal = sa.shift_stack(s, Popts, n_cols=n_cols, n_rows=n_rows, ndx=ndx, ndy=ndy, sa
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

S