PyEBSDIndex

Release 0.2.dev1

Dave Rowenhorst

Aug 08, 2023

CONTENTS

Python based tool for Radon based EBSD orientation indexing.

The pattern processing is based on a GPU pipeline, and is based on the work of S. I. Wright and B. L. Adams. Metallurgical Transactions A-Physical Metallurgy and Materials Science, 23(3):759–767, 1992, and N. C. Krieger Lassen. Automated Determination of Crystal Orientations from Electron Backscattering Patterns. PhD thesis, The Technical University of Denmark, 1994.

The band indexing is achieved through triplet voting using the methods outlined by A. Morawiec. Acta Crystallographica Section A Foundations and Advances, 76(6):719–734, 2020.

Additionally NLPAR pattern processing is included (original distribution NLPAR); P. T. Brewick, S. I. Wright, and D. J. Rowenhorst. Ultramicroscopy, 200:50–61, May 2019.).

CHAPTER

ONE

USER GUIDE

1.1 Installation

The package can be installed with pip, conda, or from source, and supports Python >= 3.7. All alternatives are available on Windows, macOS and Linux.

In order to avoid potential conflicts with other system Python packages, it is strongly recommended to use a virtual environment, such as `venv` or `conda` environments.

1.1.1 With pip

Installing all of PyEBSDIndex' functionalities with `pip`:

```
pip install pyebsdindex[all]
```

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

```
pip install pyebsdindex
```

See the following list of selectors to select the installation of optional dependencies required for specific functionality:

- `gpu` - GPU support from pyopencl. Please refer to the pyopencl installation documentation in case installation fails.
- `parallel` - Parallel indexing from ray[default].
- `all` - Install the dependencies in the above selectors.
- `doc` - Dependencies to build the documentation.
- `tests` - Dependencies to run tests.
- `dev` - Install dependencies in the above selectors.

1.1.2 With conda

GPU support is included when installing from Anaconda. On Linux or Windows:

```
conda install pyebsdindex -c conda-forge
```

On macOS (without `ray[default]`, which has to be installed separately):

```
conda install pyebsdindex-base -c conda-forge
```

1.1.3 From source

Installing the package from source with optional dependencies for running tests:

```
git clone https://github.com/USNavalResearchLaboratory/PyEBSDIndex
cd PyEBSDIndex
pip install -e .[tests]
```

Also, if you want to run the example Jupyter notebooks in the documentation, you will need to install `jupyterlab`:

```
pip install jupyterlab
```

or:

```
conda install jupyterlab
```

1.1.4 Additional installation notes

MacOS

The latest versions of `pyopencl` installed from Anaconda do not automatically include linking to the MacOS OpenCL framework. If using a `conda` environment, it may be necessary to install:

```
conda install -c conda-forge ocl_icd_wrapper_apple
```

Apple in recent installs has switched to `zsh` as the default shell. It should be noted that `zsh` sees `\[...\]` as a pattern. Thus commands like:

```
pip install pyebsdindex[gpu]
```

will return an error `"zsh: no matches found: [gpu]"`. The solution is to put the command within `'...'` such as:

```
pip install 'pyebsdindex[gpu]'
```

MacOS with Apple Silicon

The `ray` package used for distributed multi-processing only experimentally supports Apple's ARM64 architecture. More info is available here. In brief, to run on Apple ARM64, PyEBSDIndex should be installed in a `conda` environment. Assuming that `ray` has already been installed (perhaps as a dependency) and one has activated the conda environment in the terminal, run the commands below (the first two commands are to guarantee that `grpcio` is fully removed, they may send a message that the packages are not installed):

```
pip uninstall ray
pip uninstall grpcio
conda install -c conda-forge grpcio
pip install 'ray[default]'
```

1.2 Tutorials

This page contains more in-depth guides for using PyEBSDIndex. It is broken up into sections covering specific topics.

For descriptions of the functions, modules, and objects in PyEBSDIndex, see the *API reference*.

1.2.1 Indexing

Radon indexing of a demo dataset

[1]:
```
# if installed from conda or pip, this is likely not necessary, but if installed from␣
↪source, or using a developer branch, this can be quite useful.
#import sys
#sys.path.insert(0, "/Path/to/PyEBSDIndex")
```

[47]:
```
import matplotlib.pyplot as plt
import numpy as np
import h5py
import copy
from pyebsdindex import tripletvote, ebsd_pattern, ebsd_index, ebsdfile, pcopt
from pyebsdindex.EBSDImage import IPFcolor
```

An example of indexing a file of patterns.

Currently, the only types of files that can be indexed are the EDAX UP1/2 files, Oxford .ebsp uncompressed files, and HDF5 files. There are built in tools to auto-recognize h5oim and Bruker HDF5 files. Also see below on how to use h5py to input patterns from any (within some constraints) type of HDF5 file.

First we define the environmental conditions of the data collection:

[50]:
```
file = '/Path/to/example.up1' # or ebsp, or h5oina or Bruker h5
PC = np.array([0.46, 0.70, 0.64]) # this is pulled from the .ang file, but only is a␣
↪rough guess.  We will refine in a bit.
cam_elev = 5.3
sampleTilt = 70.0
vendor = 'EDAX'
```

Set up some phases. There are some shortcuts for common phases (FCC, BCC, HCP). It should be noted that the setting up of the phase information also is initializing the method used for indexing the detected bands. The default is to use triplet voting.

For the first phase, we will use the shortcut method for FCC. In its shortest form it will act as a generic FCC phase. This will automatically define the space group, set a lattice parameter = [1.0, 1.0, 1.0, 90, 90, 90], and define a set of reflecting pole families and set the phase name to "FCC".

[4]:
```
fcc = tripletvote.addphase(libtype = 'FCC' )
```

It is possible to override the defaults for any of the parameters and to set a phase name. For example:

[5]:
```
austenite = tripletvote.addphase(libtype = 'FCC', phasename = 'Austenite',␣
↪latticeparameter=[0.355, 0.355, 0.355, 90, 90, 90])
```

If the phase is not one of the shortcut phases, then the space group, lattice parameters, and reflecting families need to be defined. It should be noted that PyBSDIndex does no checks to make sure that the space group and lattice parameters have a matching specification. Thus, if hexagonal lattice parameters are input to a cubic space group, it will produce nonsense results. Here, we will use a BCC lattice as an example:

[6]:
```
ferrite = tripletvote.addphase(phasename = 'Ferrite',
                               spacegroup = 229,
                               latticeparameter=[0.286,0.286,0.286,90, 90, 90],
                               polefamilies =[[0, 1, 1], [0, 0, 2], [1, 1, 2], [0, 1, 3]])
```

Finally, we need to put these into a list. As an implementation note, the default behavior is that if PyEBSDIndex matches at least seven bands to a phase, then the second phase is not even checked. This is set as a reasonable trade-off for speed to accuracy, but can be changed if desired. Thus, putting the phase that is most likely to be found in the scan first does tend to index faster.

[7]:
```
phaselist = [austenite, ferrite]
```

For the truly lazy among us, there is also the option to define the shortcut phases as part of the list, which can be mixed and matched with the fully initiated phases above:

[8]:
```
phaselistlazy = [austenite, 'BCC', 'HCP']
```

Define the radon and indexing parameters. These work well for 60 x 60 patterns. The most critical values are the size of `rSig` and `tSig`, which are fairly dependent on the band widths.

[9]:
```
nT = 180 # 180/nTheta == degree resolution
nR = 90
tSig = 2.0 # amount of gaussian kernel size in theta in units of radon pixels.
rSig = 2.0 # amount of gassian 2nd derivate in rho in units of radon pixels.
rhomask = 0.1 # fraction of radius to not analyze
backgroundsub = False # enable/disable a simple background correction of the patterns
nbands = 8
```

Now initialize the indexer object. It is easiest to run it over 1000 patterns to give some idea of timing.
Verbose = 1 is only timing, verbose = 2 is radon and peak ID image of last pattern, verbose = 0 nothing is reported. Here, "dat1" is a temporary indexed data of the 1000 points.

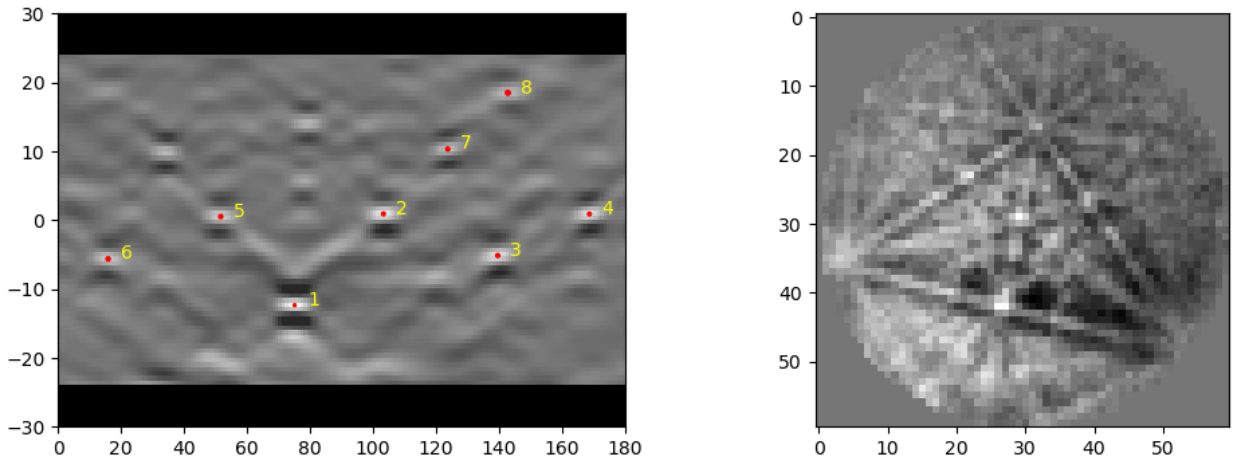
The indexer object will hold all the information needed to index a set of patterns. This includes all the environmental

conditions, the radon/band finding parameters, the phase information (including a library of triplet angles). The only parameter used are the angles between bands, no bandwidth information is currently used.

[10]:
```
dat1,bnd1, indxer=ebsd_index.index_pats(filename = file,
                                        patstart = 0, npats = 1000,return_indexer_obj =␣
↪True,
                                        backgroundSub = backgroundsub,
                                        nTheta = nT, nRho=nR,
                                        tSigma = tSig, rSigma = rSig,rhoMaskFrac=rhomask,
↪nBands=nbands, \
                                        phaselist = phaselist, PC = PC, verbose = 0)
imshape = (indxer.fID.nRows, indxer.fID.nCols)
indxer.bandDetectPlan.useCPU = False
dat1,bnd1=ebsd_index.index_pats(filename = file,
                                        patstart = 0, npats = 1000, ebsd_indexer_obj=indxer,
↪ verbose=2)
```

```
Radon Time: 0.026163205970078707
Convolution Time: 0.03730318299494684
Peak ID Time: 0.04111986805219203
Band Label Time: 0.04565136507153511
Total Band Find Time: 0.1504965319763869
Band Vote Time:  1.3719220280181617
```

```
<Figure size 640x480 with 0 Axes>
```



The data output *dat1* here, is a complex numpy array (or array of structured data), that is `[nphases+1, npoints]`. The data is stored for each phase used in indexing and the dat1[-1] layer uses the best guess on which is the most likely phase, based on the fit, and number of bands matched for each phase. Each data entry contains the orientation expressed as a quaternion (quat) (using EDAX convention by default), Pattern Quality (pq), Confidence Metric (cm), Phase ID (phase), Fit (fit) and Number of Bands Matched (nmatch). There are some other metrics reported, but these are mostly for debugging purposes.

Refine the PC guess

Here we read a set of 5x5 patterns from the center of the scan to make an optimized estimate of the pattern center. The patterns are read into a numpy array. Currently, only a single PC is used for each scan, but improvements for this should be coming soon. With the default optimization method, the initial guess should be close; within $\pm 0.1 - 0.05$, and closer is better.

[25]:
```
startcolrow = [int(imshape[1]//2)-2, int(imshape[0]//2)-2]
fID = ebsd_pattern.get_pattern_file_obj(file)
# returns patterns in an array, and the location in microns of the patterns witin the␣
↪scan relative to the center of the scan
pats, xyloc = fID.read_data(returnArrayOnly=True, convertToFloat=True,␣
↪patStartCount=[startcolrow, [5,5]])
newPC = pcopt.optimize(pats, indxer, PC0 = PC)
# actually save the PC into the indxer object.
indxer.PC = newPC
print(newPC)
```

```
[0.47484187 0.69939625 0.64461076]
```

Now use that indexer object to index the whole file. Setting `npats = -1` will index to the end of the file/array (latter on will be an example of using an array as input).

The defaults will be to detect all the GPUs on your machine, and use them. Scheduling is dynamic, so it does not matter if the GPUs are matched. After radon processing/peak finding, the cpus take over for performing the index voting – thus the number of CPUs needed will depend highly on the number of phases that need to be indexed. The number of CPUs needed also is dependent on how fast your GPUs are - on a 2019 MacPro with a Radeon 6800 GPU there are diminishing returns of including more than 32 CPUs when using the above conditions.

The first time this executes, it will take longer as the JIT compilers need to do the initial compile. Currently, the program cache is set to the system `/tmp` directory, so after reboots, many of the programs will need to be recompiled (which happens automatically with the first run)

[39]:
```
data, bnddata = ebsd_index.index_pats_distributed(filename = file,patstart = 0, npats = -
↪1,ebsd_indexer_obj = indxer,  ncpu = 28 ) #, gpu_id = [0])
```

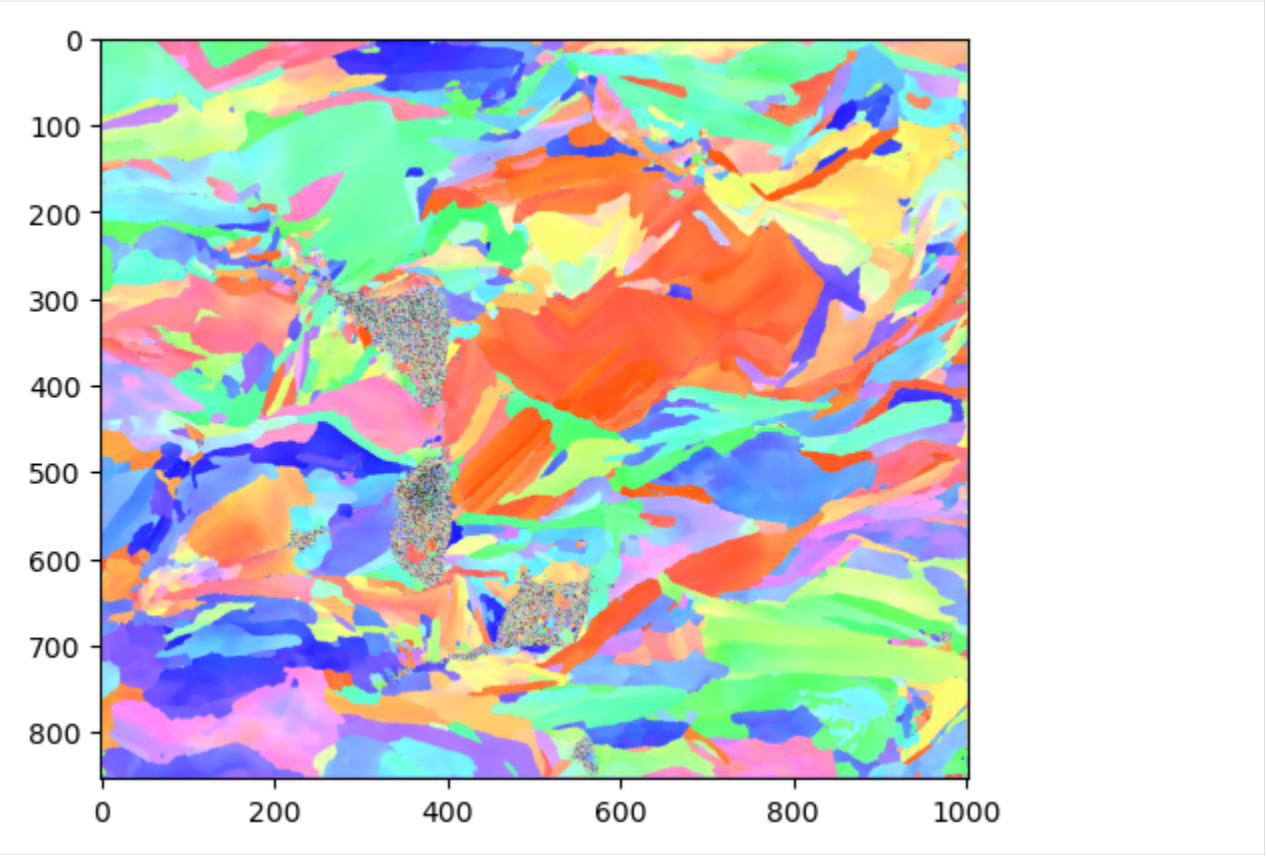
```
num cpu/gpu, and number of patterns per iteration: 28 2 1248 16 28
Completed:  853632  --  854854   PPS: 12441    100% 69;0 running;remaining(s)
```

Display the results as an IPF map. So far the only implementation is for cubic and hex IPFs - further, more flexible representations are needed here, but are likely going to be best handled by orix or others.

[27]:
```
ipfim = IPFcolor.makeipf(data, indxer, xsize = imshape[1]); plt.imshow(ipfim)
```

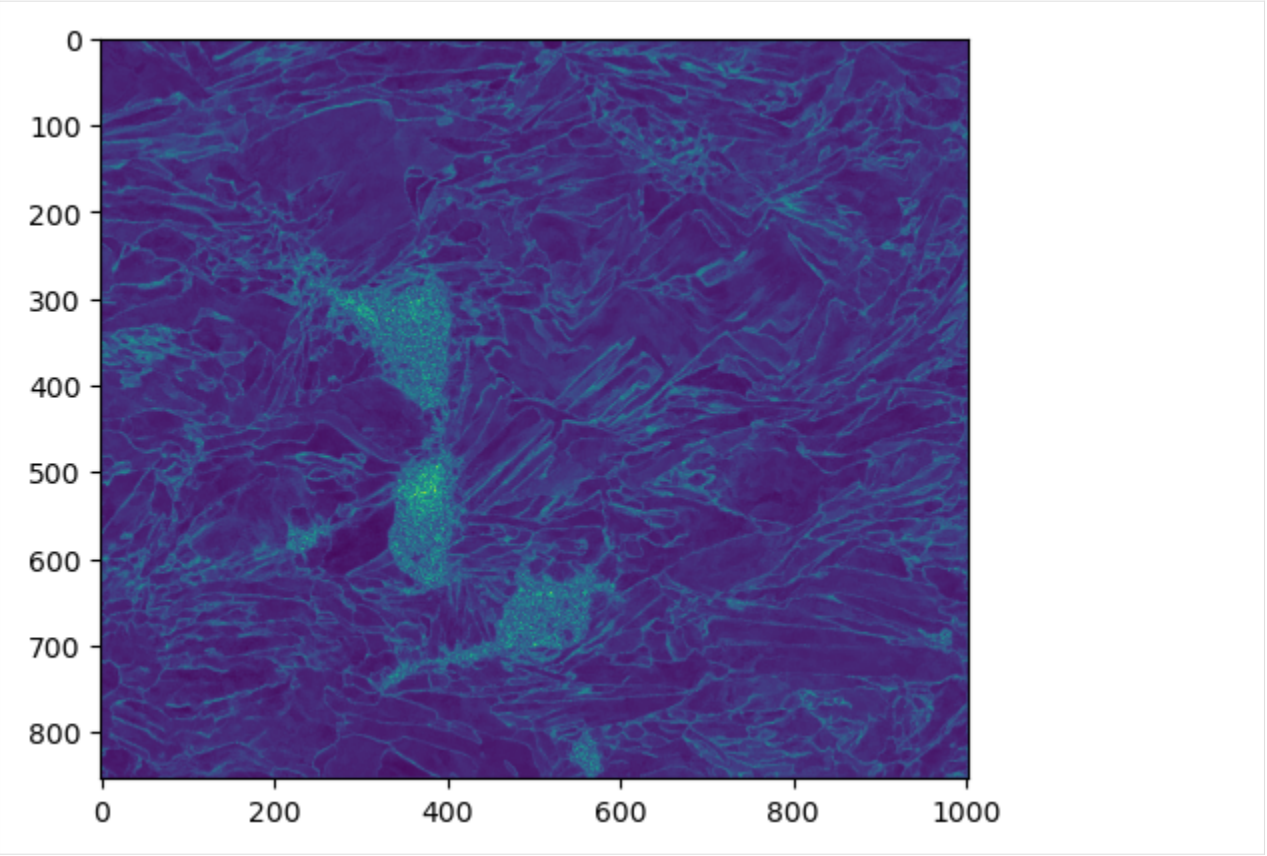
```
854
```

[27]:
```
<matplotlib.image.AxesImage at 0x2f6057400>
```



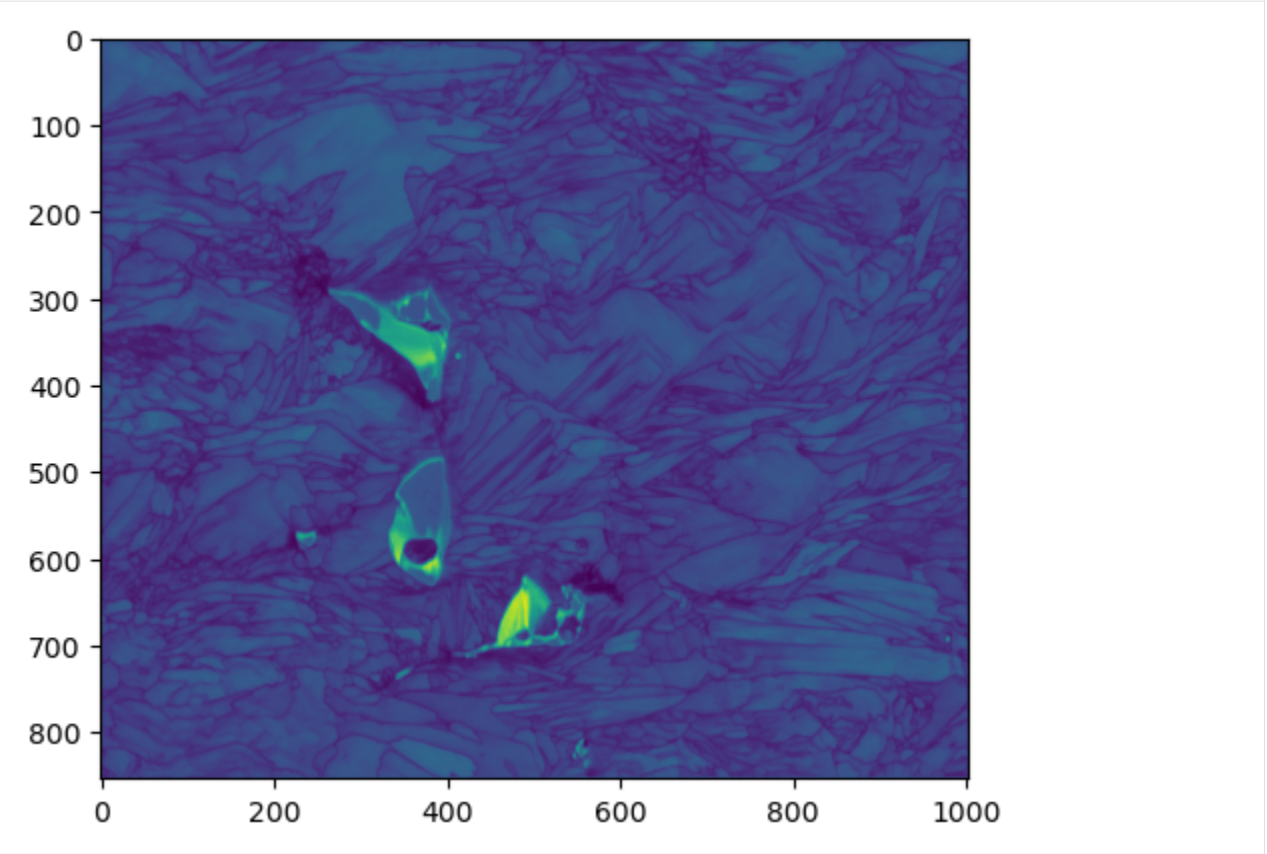
[28]:
```
fit = (data[-1]['fit']).reshape(imshape[0],imshape[1]); plt.imshow(fit.clip(0, 2.0))
```

[28]:
```
<matplotlib.image.AxesImage at 0x2f5eeb280>
```



[29]:
```
pq = (data[-1]['pq']).reshape(imshape[0],imshape[1]); plt.imshow(pq)
```

[29]:
```
<matplotlib.image.AxesImage at 0x2f79af520>
```



Writing data out

Still working on this, but there are two output formats for the data arrays from PyEBSDIndex, .ang files, and .oh5 (EDAX's version of the H5EBSD data spec).

[42]:
```
ebsdfile.writeoh5(filename='MyScanData.oh5', indexer=indxer, data=data)
```

[43]:
```
ebsdfile.writeang(filename='MyScanData.ang', indexer=indxer, data=data)
```

An example of indexing an array of patterns.

It is also possible to index a numpy array of patterns.
Here we will read part of the UP file above into an array – note that patterns can take up a lot of RAM. It is not normally advisable to read in an entire file of patterns if the file size is > 2GB.

Here we read in 200cols x 300 rows = 60000 patterns starting at column 10, row 5 (0-index based) of the EBSD scan data. What is important here is that the patterns are returned as a `(N, pH, pW)` numpy float32 array where *N* is the number of patterns, `pH` is the pattern height, and `pW` is the pattern width.

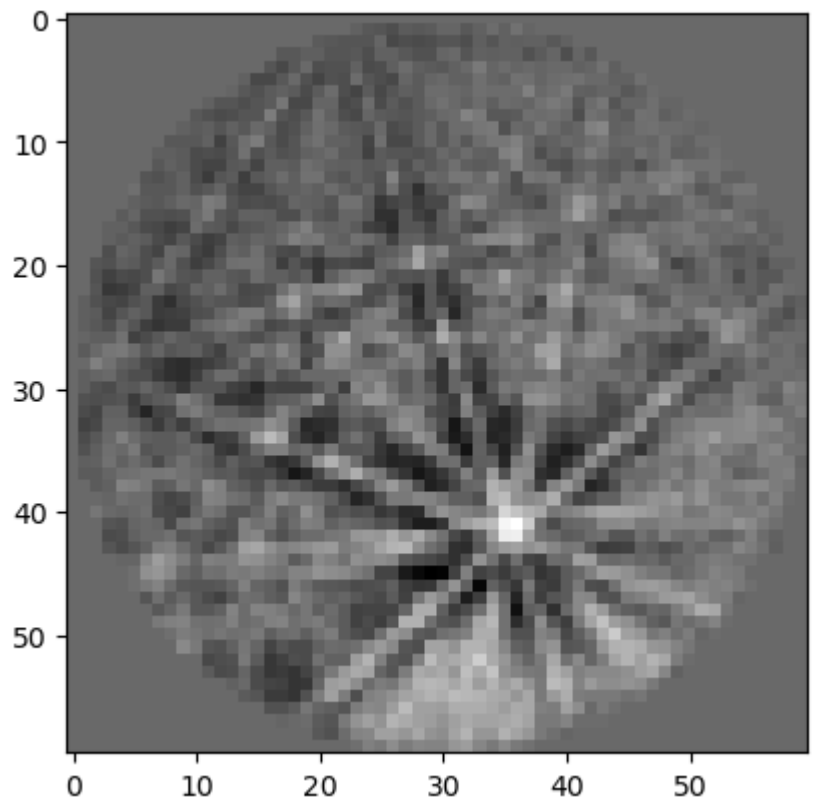
It should be noted that patterns are expected to be arranged so that `pats[0,0,0]` corresponds to the top-left pixel as one looks at the detector towards the sample (same as the EBSD vendor standards and EMSoft version >=5.0).

[30]:
```
startcolrow = [10,5]
ncol = 200
nrow = 300

f = ebsd_pattern.get_pattern_file_obj(file)
pats, xyloc = f.read_data(returnArrayOnly=True, convertToFloat=True,␣
↪patStartCount=[startcolrow, [ncol,nrow]])
# read data and return the patterns as an ndarray[npats, npatrows, npatcols], and the x,
↪y locations within the scan (in microns), ndarray[2,npats]
print(pats.shape)
print(pats.dtype)
plt.imshow(pats[0, :, :], cmap='gray')
```

```
(60000, 60, 60)
float32
```

[30]:
```
<matplotlib.image.AxesImage at 0x2f7a21ea0>
```



If the array holds a small number of patterns that can all fit on the GPU at one time, one can avoid the distributed indexing method. It should be noted that there is built in chunking (set to fairly conservative limits) to the GPU when using *index_pats*, but no multi-processing of the band voting, so it may take a long while. This small set takes about 1.5 minutes on a 2019 Mac Pro.

[32]:
```
datasm, bnddatsm, indxer=ebsd_index.index_pats(patsin = pats,
                                    return_indexer_obj = True,
                                    backgroundSub = backgroundsub,
                                    nTheta = nT, nRho=nR,
```

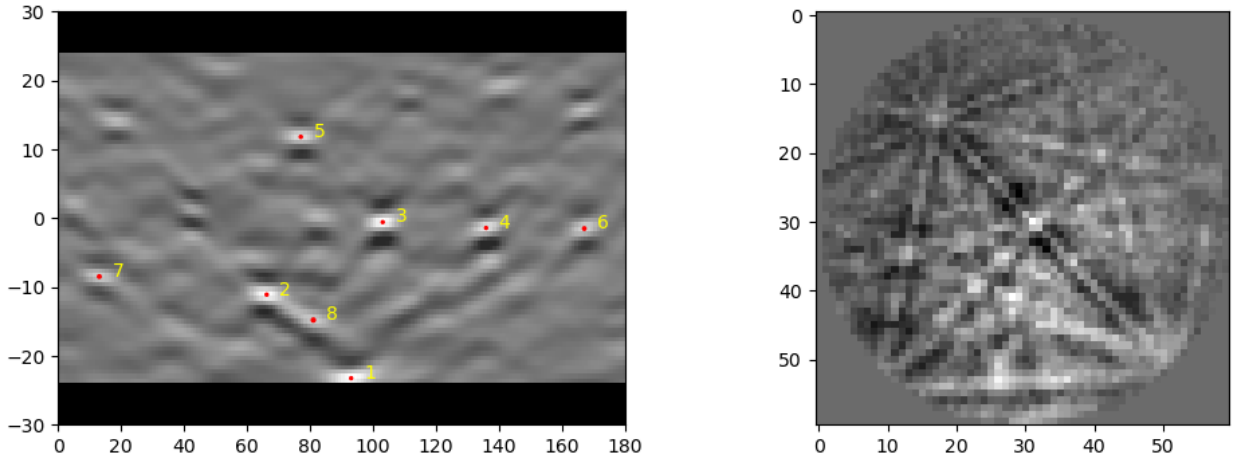
(continues on next page)

(continued from previous page)

```
                                        tSigma = tSig, rSigma = rSig,rhoMaskFrac=rhomask,
↪nBands=nbands, \
                                        phaselist = phaselist, verbose = 2)
```

```
Radon Time: 1.2478941651061177
Convolution Time: 2.1195158280897886
Peak ID Time: 1.6990612583467737
Band Label Time: 1.8449941849103197
Total Band Find Time: 6.913202239898965
Band Vote Time:  75.60198848193977
```

```
<Figure size 640x480 with 0 Axes>
```



If the array is large, then the distributed indexing works on large input arrays as well. Here a smaller number of CPU processes are used to minimize overhead of spinning up a new process.

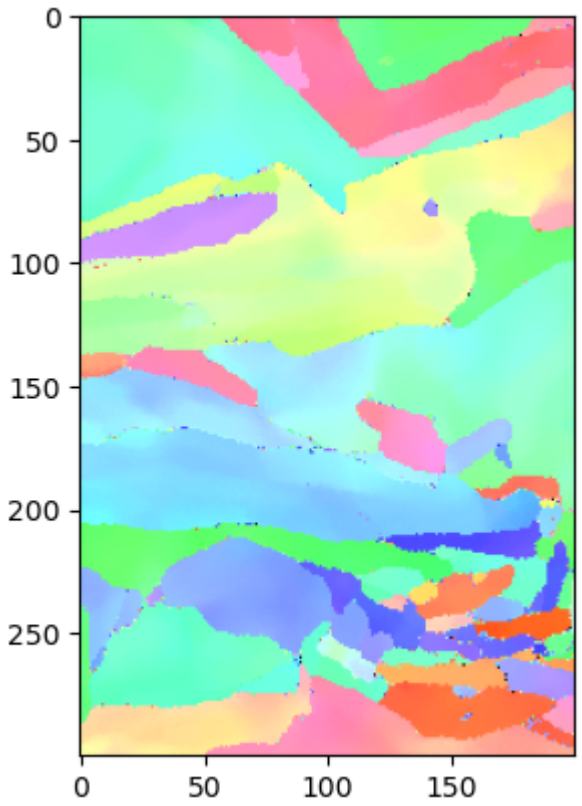
[33]:
```
datasm, bnddatsm = ebsd_index.index_pats_distributed(patsin = pats, ebsd_indexer_obj =␣
↪indxer,  ncpu = 12)
```

```
num cpu/gpu, and number of patterns per iteration: 12 2 1248 16 12
Completed:  18720  --  19968   PPS: 3716    100% 16;0 running;remaining(s)
```

[35]:
```
ipfim = IPFcolor.makeipf(datasm, indxer, xsize = 200); plt.imshow(ipfim)
```

```
300
```

[35]:
```
<matplotlib.image.AxesImage at 0x2f7cf6620>
```

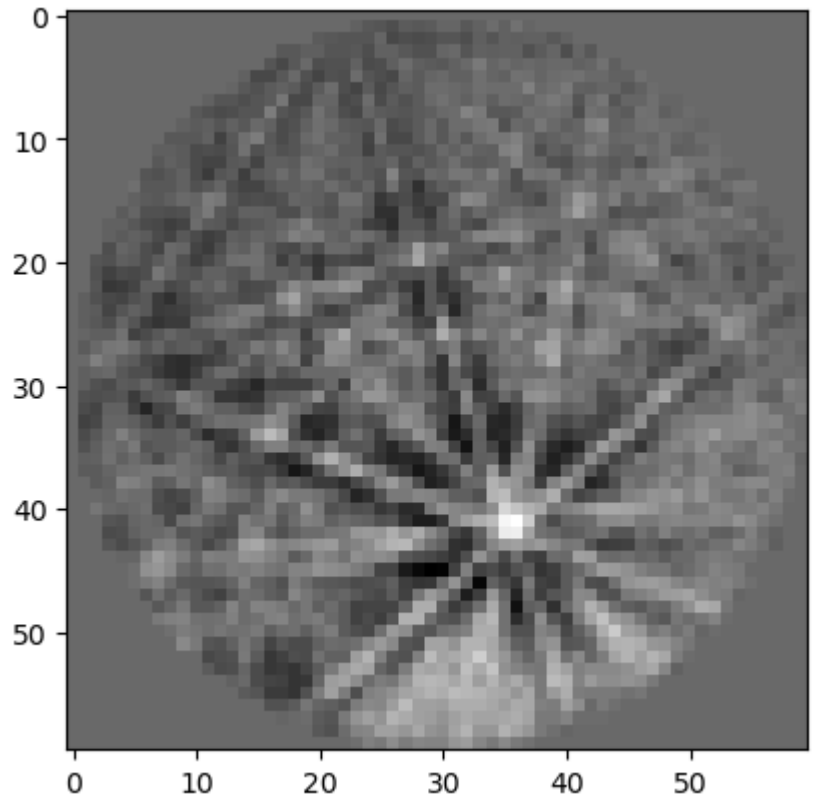


And of course, one can index a single pattern as well. In this case, *pat* can be a 2D array `(pH, pW)`:

[44]:
```
pat1 = pats[0,:, :]
print(pat1.shape)
plt.imshow(pat1, cmap='gray')
```

```
(60, 60)
```

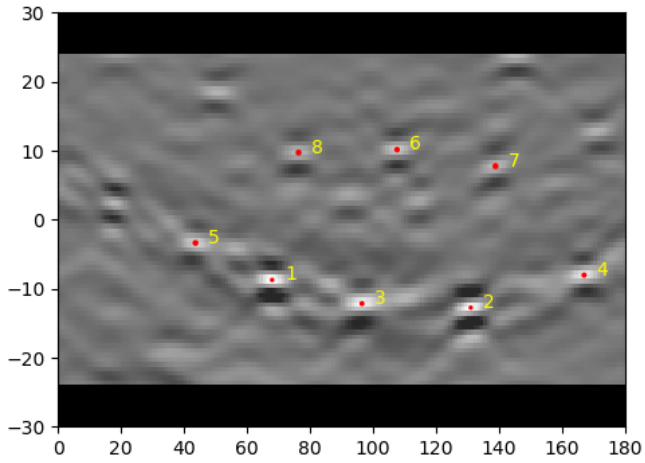
[44]:
```
<matplotlib.image.AxesImage at 0x2f7e69a80>
```

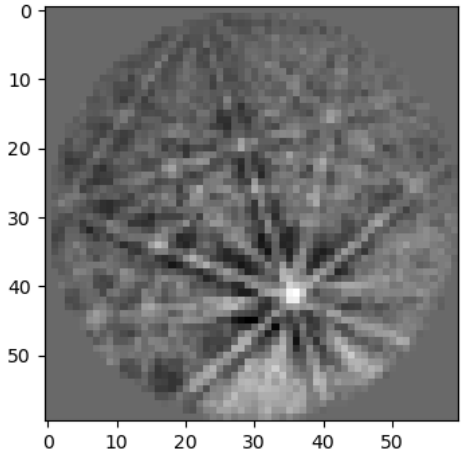


[45]:
```
dat1, bnddat1 = ebsd_index.index_pats(patsin = pat1, ebsd_indexer_obj = indxer,␣
↪verbose=2)
dat1 = dat1[-1]
print(dat1.dtype.names)
print(dat1)
```

```
Radon Time: 0.007377516012638807
Convolution Time: 0.015867227921262383
Peak ID Time: 0.01943869306705892
Band Label Time: 0.009846666012890637
Total Band Find Time: 0.0525470309657976
Band Vote Time:  0.001555563067086041
('quat', 'iq', 'pq', 'cm', 'phase', 'fit', 'nmatch', 'matchattempts', 'totvotes')
[([ 0.38395016, -0.20891693,  0.28739166,  0.85225702], 0., 158503.9, 0.77072525, 0, 0.
↪4607674, 8, [0, 1, 2, 0], 8)]
```

```
<Figure size 640x480 with 0 Axes>
```



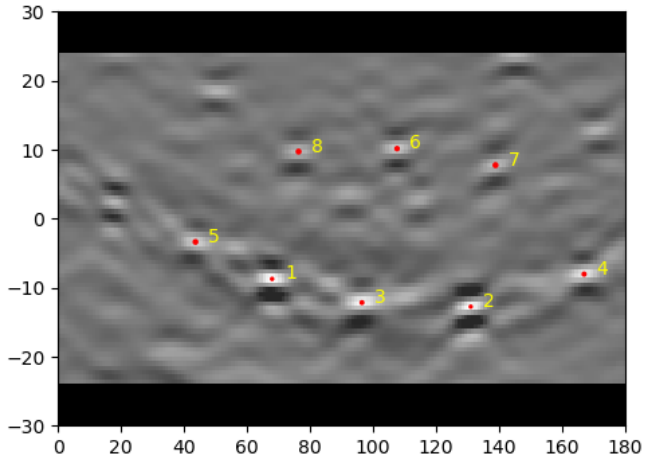


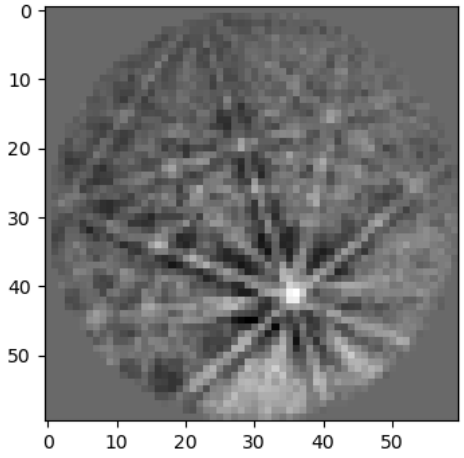
It should be noted that this is a pretty slow indexing of one point. It may be prefered to run this all on the CPU instead:

[48]:
```
indxerCPU = copy.deepcopy(indxer)
indxerCPU.bandDetectPlan.useCPU = False
dat1, bnddat1 = ebsd_index.index_pats(patsin = pat1, ebsd_indexer_obj = indxerCPU,␣
↪verbose=2)
dat1 = dat1 = dat1[-1]
```

```
Radon Time: 0.007330777938477695
Convolution Time: 0.019043672014959157
Peak ID Time: 0.026836892939172685
Band Label Time: 0.010746412095613778
Total Band Find Time: 0.06397646304685622
Band Vote Time:  0.001546425046399235
```

```
<Figure size 640x480 with 0 Axes>
```





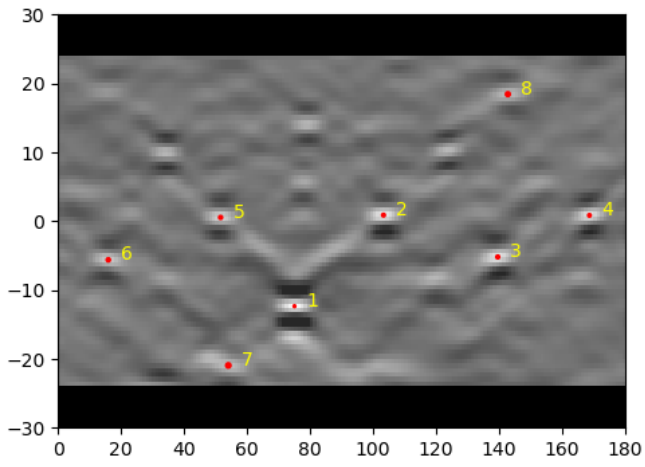
Loading data from an HDF5 File

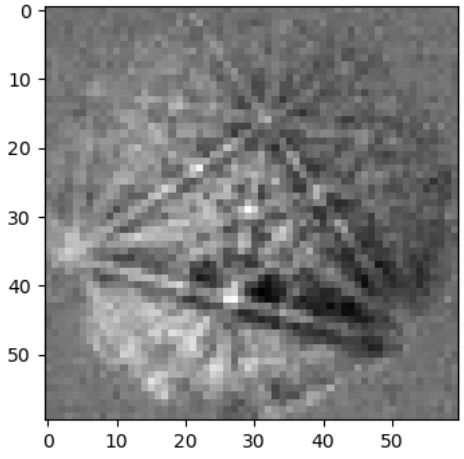
There is some limited support for specific types of HDF5 files using the *"filename"* keyword to `index_pats` or `index_pats_distributed`. However, probably the easiest method is to just point a h5py Dataset at the `patsin` keyword (This makes the important assumption that the patterns are stored in `[npatterns, nrows, ncols]` and the first point stored is the upper left corner of the detector). See below:

[49]:
```
h5file = '/Path/to/hdf5/file.h5'
f = h5py.File(h5file, 'r') # this is an HDF5 file type used by EDAX.
h5pats = f['/Scan 1/EBSD/Data/Pattern'] # location of the pattern array within the HDF5␣
↪file.
# index the first 1000
h5data, h5bnddata, indxer=ebsd_index.index_pats(patsin = h5pats[0:1000,:,:],
                                        patstart = 0, npats = 1000,return_indexer_obj =␣
↪True,
                                        backgroundSub = backgroundsub,
                                        nTheta = nT, nRho=nR,
                                        tSigma = tSig, rSigma = rSig,rhoMaskFrac=rhomask,
↪nBands=nbands, \
                                        phaselist = phaselist, PC = PC, verbose = 2)
#now index them all
h5data, h5banddata = ebsd_index.index_pats_distributed(patsin = h5pats, ebsd_indexer_obj␣
↪= indxer, ncpu = 28)
```

```
Radon Time: 0.019871829077601433
Convolution Time: 0.05477267492096871
Peak ID Time: 0.046635095961391926
Band Label Time: 0.04914216499309987
Total Band Find Time: 0.17046290694270283
Band Vote Time:  1.3323961960850284
num cpu/gpu, and number of patterns per iteration: 28 2 1008 16 28
Completed:  852768  --  853776  PPS: 12366    100% 69;0 running;remaining(s)
```

```
<Figure size 640x480 with 0 Axes>
```





[ ]:

1.2.2 Pattern processing

NLPAR Examples

[1]:
```
from pyebsdindex import nlpar
```

[2]:
```
file0 = '~/Desktop/SLMtest/scan2v3.up1'
```

[3]:
```
nlobj = nlpar.NLPAR(file0,lam=0.9, searchradius=3)
```

As always, the search radius is the 1/2 window search size. So the full window will be *(2*searchradius+1)^2* or in this case 7x7 patterns (including the center pattern of interest).

Estimating a value for lambda

A value of 0.9 is a pretty good guess for lambda for relatively noisy 80x80 patterns. But one can get a customized value of lambda by running an optimization that examines what value of lambda would provide a certain amount of reduction in the weight of the pattern of interest for a nearest-neighborhood search window. The idea being that for most scans, nearly all the neighboring patterns are nearly identical other than noise. Thus - the weigh of the pattern of interest is a measure of how much the neighboring patterns are contributing. Three optimized weights are considered (by default: [0.5, 0.34, 0.25]). Hueristically we have found that the 0.34 provides a reasonable estimate – the other two values are provided as something that represents a resonable range for lambda (lower lambda is less neighbor averaging, higher is more neighbor averaging).

Note, this will also calculate a per-point estimation of the noise in each pattern (sigma) which will be automatically stored in the nlobj for use in the NLPAR calculations.

`chunksize = int (default: 0)` These are the number of rows from the EBSD scan to process at one time. The default (set to 0) is to examine the number of columns in the scan, and estimate a chunk size that is approximately 4GB. No promises.

`automask = True (default: True)` will place a circular mask around the pattern (using the diamter of the shorter of the two pattern edges). `autoupdate = True (default: True)` will update the lambda value in the nlobj class with the optimized version.

`backsub = True (default: False)` will perform a basic background subtraction on the patterns before elvaluation. Average pattern is calculated per pattern chunk.

`saturation_protect = True (default:True)` this will exclude the pixels that have the maximum brightness value from the calculation (maximum value again calculated per pattern chunk).

[4]:
```
nlobj.opt_lambda(chunksize = 0, automask = True, autoupdate=True, backsub = False)
```

```
Chunk size set to nrows: 278
Block 0
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels␣
↪instead.
```

```
Block 278
Block 556
Block 834
Range of lambda values:  [0.65239258 0.90292969 1.15952148]
Optimal Choice:  0.9029296874999998
```

Executing NLPAR

Now that there are reasonable estimates for sigma and lambda, one can execute NLPAR. With the default values and using 'UP' EDAX files, a new file will be created to store the result with the pattern [filename]lam[x.xx]sr[x]dt[x.x].up[1/2].

The user can override the filename if they want - but should not overwrite the original data filename (no protections are provided).

[6]:
```
nlobj.searchradius = 4
nlobj.calcnlpar(chunksize=0,searchradius=None,lam = None, saturation_protect=True,
↪automask=True,
                filename=None, fileout=None, backsub = False)
```

```
Chunk size set to nrows: 278
0.90292966 4 0.0
Block 0
Block 278
Block 556
Block 834
```

[5]:
```
nlobj.calcnlpar(chunksize=0,searchradius=None,lam = None, saturation_protect=True,
↪automask=True,
                filename=None, fileout='/tmp/dave.up1', backsub = False)
```

```
Chunk size set to nrows: 278
0.90292966 3 0.0
Block 0
Block 278
Block 556
Block 834
```

[6]:
```
nlobj.calcnlpar(chunksize=0,searchradius=11,lam = 1.2, saturation_protect=True,
↪automask=True,
                filename=None, fileout='/tmp/dave2.up1', backsub = False)
```

```
Chunk size set to nrows: 278
1.2 11 0.0
Block 0
Block 278
Block 556
Block 834
```

[ ]:

CHAPTER

TWO

API REFERENCE

Release: 0.2.dev1

Date: Aug 08, 2023

This reference manual details the public functions, modules, and objects in PyEBSDIndex. For learning how to use PyEBSDIndex, see the *Tutorials*.

Caution: PyEBSDIndex is in development. This means that some breaking changes and changes to this reference are likely with each release.

Functionality is inteded to be imported like this:

```
>>> from pyebsdindex import ebsd_index, pcopt
>>> indexer = ebsd_index.EBSDIndexer()
```

Modules

ebsd_index	Setup and handling of Radon indexing runs of EBSD patterns.
nlpar	Non-local pattern averaging and re-indexing (NLPAR).
pcopt	Optimization of the pattern center (PC) of EBSD patterns.
tripletvote	Creation of look-up tables from phase information for band indexing.

2.1 ebsd_index

Setup and handling of Radon indexing runs of EBSD patterns.

Functions

index_pats([patsin, filename, phaselist, ...])	Index EBSD patterns on a single thread.

2.1.1 index_pats

pyebsdindex.ebsd_index.**index_pats**(*patsin=None*, *filename=None*, *phaselist=['FCC']*, *vendor=None*, *PC=None*, *sampleTilt=70.0*, *camElev=5.3*, *bandDetectPlan=None*, *nRho=90*, *nTheta=180*, *tSigma=None*, *rSigma=None*, *rhoMaskFrac=0.1*, *nBands=9*, *backgroundSub=False*, *patstart=0*, *npats=-1*, *return_indexer_obj=False*, *ebsd_indexer_obj=None*, *clparams=None*, *verbose=0*, *chunksize=528*, *gpu_id=None*)

Index EBSD patterns on a single thread.

Parameters

patsin
[numpy.ndarray, optional] EBSD patterns in an array of shape (n points, n pattern rows, n pattern columns). If not given, these are read from `filename`.

filename
[str, optional] Name of file with EBSD patterns. If not given, `patsin` must be passed.

phaselist
[list of str, optional] Options are `"FCC"` and `"BCC"`. Default is `["FCC"]`.

vendor
[str, optional] Which vendor convention to use for the pattern center (PC) and the returned orientations. The available options are `"EDAX"` (default), `"BRUKER"`, `"OXFORD"`, `"EMSOFT"`, `"KIKUCHIPY"`.

PC
[list, optional] Pattern center (PCx, PCy, PCz) in the `indexer.vendor` or `vendor` convention. For EDAX TSL, this is (x*, y*, z*), defined in fractions of pattern width with respect to the lower left corner of the detector. If not passed, this is set to (x*, y*, z*) = (0.471659, 0.675044, 0.630139). If `vendor="EMSOFT"`, the PC must be four numbers, the final number being the pixel size.

sampleTilt
[float, optional] Sample tilt towards the detector in degrees. Default is 70 degrees. Unused if `ebsd_indexer_obj` is passed.

camElev
[float, optional] Camera elevation in degrees. Default is 5.3 degrees. Unused if `ebsd_indexer_obj` is passed.

bandDetectPlan
[pyebsdindex.band_detect.BandDetect, optional] Collection of parameters using in band detection. Unused if `ebsd_indexer_obj` is passed.

nRho
[int, optional] Default is 90 degrees. Unused if `ebsd_indexer_obj` is passed.

nTheta
[int, optional] Default is 180 degrees. Unused if `ebsd_indexer_obj` is passed.

tSigma
[float, optional] Unused if ebsd_indexer_obj is passed.

rSigma
[float, optional] Unused if ebsd_indexer_obj is passed.

rhoMaskFrac
[float, optional] Default is 0.1. Unused if ebsd_indexer_obj is passed.

nBands
[int, optional] Number of detected bands to use in triplet voting. Default is 9. Unused if ebsd_indexer_obj is passed.

backgroundSub
[bool, optional] Whether to subtract a static background prior to indexing. Default is False.

patstart
[int, optional] Starting index of the patterns to index. Default is 0.

npats
[int, optional] Number of patterns to index. Default is -1, which will index up to the final pattern in patsin.

return_indexer_obj
[bool, optional] Whether to return the EBSD indexer. Default is False.

ebsd_indexer_obj
[*EBSDIndexer*, optional] EBSD indexer. If not given, many of the above parameters must be passed. Otherwise, these parameters are retrieved from this indexer.

clparams
[list, optional] OpenCL parameters passed to pyopencl if the package is installed.

verbose
[int, optional] 0 - no output (default), 1 - timings, 2 - timings and the Radon transform of the first pattern with detected bands highlighted.

chunksize
[int, optional] Default is 528.

gpu_id
[int, optional] ID of GPU to use if pyopencl is installed.

Returns

indxData
[numpy.ndarray] Complex numpy array (or array of structured data), that is [nphases + 1, npoints]. The data is stored for each phase used in indexing and the indxData[-1] layer uses the best guess on which is the most likely phase, based on the fit, and number of bands matched for each phase. Each data entry contains the orientation expressed as a quaternion (quat) (using the convention of vendor or indexer.vendor), Pattern Quality (pq), Confidence Metric (cm), Phase ID (phase), Fit (fit) and Number of Bands Matched (nmatch). There are some other metrics reported, but these are mostly for debugging purposes.

bandData
[numpy.ndarray] Band identification data from the Radon transform. Stored as a structured numpy array, of dimensions [npoints, nbands].

With fields that include:

- id: band ID

- max: peak max intesensity (used to calculate pattern quality)
- maxloc: nearest integer location of the Radon peak
- avemax: nearest neighbor average of the max peak intensity
- aveloc: sub-pixel location of the Radon peak
- width: a metric of the band width
- theta: the theta value of the sub-pixel location on the Radon (lower-left origin)
- rho: the rho value of the sub-pixel location on the Radon (lower-left origin)
- valid: was the peak detected
- band_match_index: index for phase number and pole number that indexed to this band (use *getmatchedpole()*)

indexer
[*EBSDIndexer*] EBSD indexer, returned if `return_indexer_obj=True`.

Classes

EBSDIndexer([filename, phaselist, vendor, ...])	Setup of Radon indexing of EBSD patterns.

2.1.2 EBSDIndexer

class pyebsdindex.ebsd_index.**EBSDIndexer**(*filename=None*, *phaselist=['FCC']*, *vendor=None*, *PC=None*, *sampleTilt=70.0*, *camElev=5.3*, *bandDetectPlan=None*, *nRho=90*, *nTheta=180*, *tSigma=1.0*, *rSigma=1.2*, *rhoMaskFrac=0.15*, *nBands=9*, *patDim=None*, *nband_earlyexit=None*, ***kwargs*)

Bases: `object`

Setup of Radon indexing of EBSD patterns.

Parameters

filename
[`str`, `optional`] Name of file with EBSD patterns.

phaselist
[`list of str`, `optional`] Options are `"FCC"` and `"BCC"`. Default is `["FCC"]`.

vendor
[`str`, `optional`] Which vendor convention to use for the pattern center (PC) and the returned orientations. The available options are `"EDAX"` (default), `"BRUKER"`, `"OXFORD"`, `"EMSOFT"`, `"KIKUCHIPY"`.

PC
[`list`, `optional`] Pattern center (PCx, PCy, PCz) in the `vendor` convention. For EDAX TSL, this is (x*, y*, z*), defined in fractions of pattern width with respect to the lower left corner of the detector. If not passed, this is set to (x*, y*, z*) = (0.471659, 0.675044, 0.630139). If `vendor="EMSOFT"`, the PC must be four numbers, the final number being the pixel size.

sampleTilt
[float, optional] Sample tilt towards the detector in degrees. Default is 70 degrees. Unused if ebsd_indexer_obj is passed.

camElev
[float, optional] Camera elevation in degrees. Default is 5.3 degrees. Unused if ebsd_indexer_obj is passed.

bandDetectPlan
[pyebsdindex.band_detect.BandDetect, optional] Collection of parameters using in band detection. Unused if ebsd_indexer_obj is passed.

nRho
[int, optional] Default is 90 degrees. Unused if ebsd_indexer_obj is passed.

nTheta
[int, optional] Default is 180 degrees. Unused if ebsd_indexer_obj is passed.

tSigma
[float, optional] Unused if ebsd_indexer_obj is passed.

rSigma
[float, optional] Unused if ebsd_indexer_obj is passed.

rhoMaskFrac
[float, optional] Default is 0.1. Unused if ebsd_indexer_obj is passed.

nBands
[int, optional] Number of detected bands to use in triplet voting. Default is 9. Unused if ebsd_indexer_obj is passed.

patDim
[int, optional] Number of dimensions of pattern array.

****kwargs**
Keyword arguments passed on to BandDetect.

Methods

EBSDIndexer.getmatchedpole(banddata[, float_out])	Return the pole from the library that was matched to the detected band.
EBSDIndexer.index_pats([patsin, patstart, ...])	Index EBSD patterns.
EBSDIndexer.update_file([filename, patDim])	Update file with patterns to index.

getmatchedpole

EBSDIndexer.**getmatchedpole**(*banddata*, *float_out=False*)

Return the pole from the library that was matched to the detected band.

Parameters

banddata
[numpy.ndarray] Output structured band data array from *index_pats()* or index_pats_distributed().

float_out
[bool, optional] Default (False) is to return an array of ints with Miller indices. If set

to True, then floats, with unit length, will be returned in the sample Cartesian reference frame. (Length is only valid for cubic systems).

Returns

numpy.ndarray
The default is an array [npoints, nbands, 3] that contains the Miller indices (as ints) of the matching pole (note that hexagonal will also return only three-index notation). If the float_out is set to True, then the output will be floating point vectors of length one, within the sample Cartesian reference frame.

index_pats

EBSDIndexer.**index_pats**(*patsin=None*, *patstart=0*, *npats=-1*, *xyloc=None*, *clparams=None*, *PC=None*, *verbose=0*, *chunksize=512*)

Index EBSD patterns.

Parameters

patsin
[numpy.ndarray, optional] EBSD patterns in an array of shape (n points, n pattern rows, n pattern columns). If not given, these are read from `self.filename`.

patstart
[int, optional] Starting index of the patterns to index. Default is `0`.

npats
[int, optional] Number of patterns to index. Default is `-1`, which will index up to the final pattern in `patsin`.

clparams
[list, optional] OpenCL parameters passed to `pyopencl`.

PC
[list, optional] Pattern center (PC) parameters (PCx, PCy, PCz) in the vendor convention. For EDAX TSL, this is (x*, y*, z*), defined in fractions of pattern width with respect to the lower left corner of the detector. If not given, this is read from `self.PC`. If `vendor` is `"EMSOFT"`, the PC must be four numbers, the final number being the pixel size.

verbose
[int, optional] 0 - no output (default), 1 - timings, 2 - timings and the Radon transform of the first pattern with detected bands highlighted.

chunksize
[int, optional] Default is 512.

Returns

indxData
[numpy.ndarray] Structured numpy array, that is [nphases + 1, npoints]. The data is stored for each phase used in indexing and the `indxData[-1]` layer uses the best guess on which is the most likely phase, based on the fit, and number of bands matched for each phase. Each data entry contains the orientation expressed as a quaternion ('quat') (using the convention of `vendor` or `indexer.vendor`), Pattern Quality ('pq'), Confidence Metric ('cm'), Phase ID ('phase'), Fit ('fit') and Number of Bands Matched ('nmatch'). There are some other metrics reported, but these are mostly for debugging purposes. The number and order of fields are not guaranteed to remain the same, but fields listed here are stable. (phase) parameter will be set to -1 for any no-solution point.

bandData
[numpy.ndarray] Band identification data from the Radon transform. Stored as a structured numpy array, of dimensions [npoints, nbands].

With fields that include:

- id: band ID
- max: peak max intesensity (used to calculate pattern quality)
- maxloc: nearest integer location of the Radon peak
- avemax: nearest neighbor average of the max peak intensity
- aveloc: sub-pixel location of the Radon peak
- width: a metric of the band width
- theta: the theta value of the sub-pixel location on the Radon (lower-left origin)
- rho: the rho value of the sub-pixel location on the Radon (lower-left origin)
- valid: was the peak detected
- band_match_index: index for phase number and pole number that indexed to this band (use *getmatchedpole()*)

patstart
[int] Starting index of the indexed patterns.

npats
[int] Number of patterns indexed. This and *patstart* are useful for the distributed indexing procedures.

update_file

EBSDIndexer.**update_file**(*filename=None*, *patDim=array([120, 120], dtype=int32)*)

Update file with patterns to index.

Parameters

filename
[str, optional] Name of file with EBSD patterns.

patDim
[numpy.ndarray] 1D array with two values, the pattern height and width.

2.2 nlpar

Non-local pattern averaging and re-indexing (NLPAR).

Classes

NLPAR([filename, lam, searchradius, ...])	

2.2.1 NLPAR

class pyebsdindex.nlpar.**NLPAR**(*filename=None*, *lam=0.7*, *searchradius=3*, *dthresh=0.0*, *nrows=None*, *ncols=None*)

Bases: object

Methods

NLPAR.automask(h, w)	
NLPAR.backsub(data)	
NLPAR.calcnlpar([chunksize, searchradius, ...])	
NLPAR.calcsigma([chunksize, nn, ...])	
NLPAR.getinfileobj()	
NLPAR.getoutfileobj()	
NLPAR.nlpar_nb(data, lam, sr, dthresh, ...)	
NLPAR.opt_lambda([chunksize, ...])	
NLPAR.setfile([filepath])	
NLPAR.setoutfile(patternfile[, filepath])	Set the output file.
NLPAR.sigma_numba(data, nn, nrows, ncols, ...)	

automask

static NLPAR.**automask**(*h*, *w*)

backsub

NLPAR.**backsub**(*data*)

calcnlpar

NLPAR.**calcnlpar**(*chunksize=0*, *searchradius=None*, *lam=None*, *dthresh=None*, *saturation_protect=True*, *automask=True*, *filename=None*, *fileout=None*, *reset_sigma=True*, *backsub=False*, *rescale=False*)

calcsigma

NLPAR.**calcsigma**(*chunksize=0*, *nn=1*, *saturation_protect=True*, *automask=True*)

getinfileobj

NLPAR.**getinfileobj**()

getoutfileobj

NLPAR.**getoutfileobj**()

nlpar_nb

static NLPAR.**nlpar_nb**(*data*, *lam*, *sr*, *dthresh*, *sigma*, *nrows*, *ncols*, *indices*, *saturation_protect=True*)

opt_lambda

NLPAR.**opt_lambda**(*chunksize=0*, *saturation_protect=True*, *automask=True*, *backsub=False*, *target_weights=[0.5, 0.34, 0.25]*, *dthresh=0.0*, *autoupdate=True*)

setfile

NLPAR.**setfile**(*filepath=None*)

setoutfile

NLPAR.**setoutfile**(*patternfile*, *filepath=None*)

Set the output file.

Parameters

patternfile
Input pattern file object from ebsd_pattern.

filepath
String.

Notes

In the future I want to be able to specify the HDF5 data path to store the output data, but that is proving to be a bit of a mess. For now, a copy of the original HDF5 is made, and the NLPAR patterns will be overwritten on top of the originals.

sigma_numba

static NLPAR.**sigma_numba**(*data*, *nn*, *nrows*, *ncols*, *rowstartcount*, *colstartcount*, *indices*, *saturation_protect=True*)

2.3 pcopt

Optimization of the pattern center (PC) of EBSD patterns.

Functions

optimize(pats, indexer[, PC0, batch])	Optimize pattern center (PC) (PCx, PCy, PCz) in the convention of the `indexer.vendor` with Nelder-Mead.
optimize_pso(pats, indexer[, PC0, batch, ...])	Optimize pattern center (PC) (PCx, PCy, PCz) in the convention of the `indexer.vendor` with particle swarms.

2.3.1 optimize

pyebsdindex.pcopt.**optimize**(*pats*, *indexer*, *PC0=None*, *batch=False*)

Optimize pattern center (PC) (PCx, PCy, PCz) in the convention of the `indexer.vendor` with Nelder-Mead.

Parameters

pats
[numpy.ndarray] EBSD pattern(s), of shape `(n detector rows, n detector columns)`, or `(n patterns, n detector rows, n detector columns)`.

indexer
[*pyebsdindex.ebsd_index.EBSDIndexer*] EBSD indexer instance storing all relevant parameters for band detection.

PC0
[list, `optional`] Initial guess of PC. If not given, `indexer.PC` is used. If `indexer.vendor` is `"EMSOFT"`, the PC must be four numbers, the final number being the pixel size.

batch
[bool, `optional`] Default is `False` which indicates the fit for a set of patterns should be optimized using the cumulative fit for all the patterns, and one PC will be returned. If `True`, then an optimization is run for each individual pattern, and an array of PC values is returned.

Returns

numpy.ndarray
Optimized PC.

Notes

SciPy's Nelder-Mead minimization function is used with a tolerance `fatol=0.00001` between each iteration, ending the optimization when the improvement is below this value.

2.3.2 optimize_pso

pyebsdindex.pcopt.**optimize_pso**(*pats*, *indexer*, *PC0=None*, *batch=False*, *search_limit=0.2*, *nswarmparticles=30*, *pswarmpar=None*, *niter=50*, *verbose=1*)

Optimize pattern center (PC) (PCx, PCy, PCz) in the convention of the `indexer.vendor` with particle swarms.

Parameters

pats
[numpy.ndarray] EBSD pattern(s), of shape `(n detector rows, n detector columns)`, or `(n patterns, n detector rows, n detector columns)`.

indexer
[*pyebsdindex.ebsd_index.EBSDIndexer*] EBSD indexer instance storing all relevant parameters for band detection.

PC0
[list, optional] Initial guess of PC. If not given, `indexer.PC` is used. If `indexer.vendor` is `"EMSOFT"`, the PC must be four numbers, the final number being the pixel size.

batch
[bool, optional] Default is `False` which indicates the fit for a set of patterns should be optimized using the cumulative fit for all the patterns, and one PC will be returned. If `True`, then an optimization is run for each individual pattern, and an array of PC values is returned.

search_limit
[float, optional] Default is 0.2 for all PC values, and sets the +/- limit for the optimization search.

nswarmparticles
[int, optional] Number of particles in a swarm. Default is 30.

pswarmpar
[dict, optional] Particle swarm parameters "c1", "c2", and "w" with defaults 3.5, 3.5, and 0.8, respectively.

niter
[int, optional] Number of iterations. Default is 50.

verbose
[int, optional] Whether to print the parameters and progress of the optimization (>= 1) or not (< 1). Default is to print.

Returns

numpy.ndarray
Optimized PC.

2.4 tripletvote

Creation of look-up tables from phase information for band indexing.

Functions

addphase([libtype, phasename, spacegroup, ...])	Return a band indexer for a phase.

2.4.1 addphase

pyebsdindex.tripletvote.**addphase**(*libtype=None*, *phasename=None*, *spacegroup=None*, *latticeparameter=None*, *polefamilies=None*, *nband_earlyexit=10*)

Return a band indexer for a phase.

Parameters

libtype
[str, optional] Shorthand definition of a phase. Options are FCC, BCC, or HCP.

phasename
[str, optional] Phase name.

spacegroup
[int, optional] Space group of the phase.

latticeparameter
[np.ndarray, tuple, or list, optional] Lattice parameters (a, b, c, alpha, beta, gamma).

polefamilies
[np.ndarray, tuple, or list, optional] Reflector families to use in indexing.

nband_earlyexit
[int, optional] If this phase is first in a list of phases used in indexing, and if this many bands are matched, the remaining phases in the list will not be checked. Default is 10, unless libtype is passed, in which case it is 8.

Returns

BandIndexer
Band indexer for this phase.

Classes

BandIndexer([phasename, spacegroup, ...])	

2.4.2 BandIndexer

class pyebsdindex.tripletvote.**BandIndexer**(*phasename=None*, *spacegroup=None*, *latticeparameter=None*, *polefamilies=None*, *angTol=2.0*, *nband_earlyexit=8*)

Bases: object

Methods

BandIndexer.bandindex(band_norms[, ...])
BandIndexer.build_trip_lib()
BandIndexer.pairVoteOrientation(bandnormsI
BandIndexer.setlatticeparameter(latticeparam
BandIndexer.setpolefamilies(reflectors)
BandIndexer.setspacegroup([spacegroup])

bandindex

BandIndexer.**bandindex**(*band_norms*, *band_intensity=None*, *band_widths=None*, *verbose=0*)

build_trip_lib

BandIndexer.**build_trip_lib**()

pairVoteOrientation

BandIndexer.**pairVoteOrientation**(*bandnormsIN*, *goNumba=True*)

setlatticeparameter

BandIndexer.**setlatticeparameter**(*latticeparameter*)

setpolefamilies

BandIndexer.**setpolefamilies**(*reflectors*)

setspacegroup

BandIndexer.**setspacegroup**(*spacegroup=225*)

CHAPTER

THREE

CONTRIBUTING

PyEBSDIndex is a community maintained project. We welcome contributions in the form of bug reports, documentation, code, feature requests, and more. The source code is hosted on GitHub. This guide provides some tips on how to build documentation and run tests when contributing.

3.1 Building and writing documentation

We use Sphinx for documenting functionality. Install necessary dependencies to build the documentation:

```
pip install --editable .[doc]
```

Then, build the documentation from the `doc` directory:

```
cd doc
make html
```

The documentation's HTML pages are built in the `doc/build/html` directory from files in the reStructuredText (reST) plaintext markup language. They should be accessible in the browser by typing `file:///your/absolute/path/to/pyebsdindex/doc/build/html/index.html` in the address bar.

Tips for writing Jupyter Notebooks that are meant to be converted to reST text files by nbsphinx:

- Use `_ = ax[0].imshow(...)` to disable Matplotlib output if a Matplotlib command is the last line in a cell.
- Refer to our reference with this general MD `[pcopt.optimize()](../reference.rst#pyebsdindex.pcopt.optimize)`. Remember to add the parentheses `()` for functions and methods.
- Refer to external references via standard MD like `[Signal2D](http://hyperspy.org/hyperspy-doc/current/api/hyperspy._signals.signal2d.html)`.
- The Sphinx gallery thumbnail used for a notebook is set by adding the `nbsphinx-thumbnail` tag to a code cell with an image output. The notebook must be added to the gallery in the *index.rst* to be included in the documentation pages.
- The PyData Sphinx theme displays the documentation in a light or dark theme, depending on the browser/OS setting. It is important to make sure the documentation is readable with both themes. This means displaying all figures with a white background for axes labels and ticks and figure titles etc. to be readable.

3.2 Running and writing tests

Some functionality in PyEBSDIndex is tested via the pytest framework. The tests reside in the `pyebsdindex/tests` directory. Tests are short methods that call functions in PyEBSDIndex and compare resulting output values with known answers. Install necessary dependencies to run the tests:

```
pip install --editable .[tests]
```

Some useful fixtures, like a dynamically simulated Al pattern, are available in the `conftest.py` file.

To run the tests:

```
pytest --cov --pyargs pyebsdindex
```

The `--cov` flag makes coverage.py print a nice report in the terminal. For an even nicer presentation, you can use `coverage.py` directly:

```
coverage html
```

Then, you can open the created `htmlcov/index.html` in the browser and inspect the coverage in more detail.

To run only a specific test function or class, .e.g the `TestEBSDIndexer` class:

```
pytest -k TestEBSDIndexer
```

This is useful when you only want to run a specific test and not the full test suite, e.g. when you're creating or updating a test. But remember to run the full test suite before pushing!

Tips for writing tests of Numba decorated functions:

- A Numba decorated function `numba_func()` is only covered if it is called in the test as `numba_func.py_func()`.
- Always test a Numba decorated function calling `numba_func()` directly, in addition to `numba_func.py_func()`, because the machine code function might give different results on different OS with the same Python code.

3.3 Continuous integration (CI)

We use GitHub Actions to ensure that PyEBSDIndex can be installed on macOS and Linux (Ubuntu). After a successful installation of the package, the CI server runs the tests. Add "[skip ci]" to a commit message to skip this workflow on any commit to a pull request.

CHAPTER

FOUR

CHANGELOG

All notable changes to PyEBSDIndex will be documented in this file. The format is based on Keep a Changelog.

4.1 0.2.dev1

4.1.1 Added

4.1.2 Changed

4.1.3 Removed

4.1.4 Fixed

4.2 0.2.0 (2023-08-08)

4.2.1 Added

- Initial support for uncompressed EBSP files from Oxford systems.
- Significant improvement in the particle swarm optimization for pattern center optimization.
- Initial support for non-cubic phases. Hexagonal verified with EDAX convention. Others are untested.
- Significant improvements in phase differentiation.
- NLPAR support for Oxford HDF5 and EBSP.
- Initial support for Oxford .h5oina files
- Added IPF coloring/legends for hexagonal phases
- Data output files in .ang and EDAX .oh5 files
- Explicit support for Python 3.11.

4.2.2 Changed

- CRITICAL! All `ebsd_pattern.EBSDPatternFiles.read_data()` calls will now return TWO arguments. The patterns (same as previous), and an nd.array of the x,y location within the scan of the patterns. The origin is the center of the scan, and reported in microns.
- `ebsd_index.index_pats_distributed()` now will auto optimize the number of patterns processed at a time depending on GPU capability, and is set as the default.
- Updated tutorials for new features.

4.2.3 Removed

- Removed requirement for installation of pyswarms.
- Removed any references to np.floats and replaced with float() or np.float32/64.

4.2.4 Fixed

- Radon transform figure when `verbose=2` is passed to various indexing methods is now plotted in its own figure.
- Several bug fixes with NLPAR file reading/writing.
- Complete rewrite of the scheduling for `ebsd_index.index_pats_distributed()` function to be compatible with NVIDIA cards.

4.3 0.1.1 (2022-10-25)

4.3.1 Added

- Explanation that the pixel size must be passed as the forth PC value whenever `vendor=EMSOFT` is used.

4.3.2 Changed

- Changed the parameter name `patsIn` to `patsin` in functions `index_pats()` and `index_pats_distributed()`, to be in line with `EBSDIndex.index_pats()`, and `peakDetectPlan` to `bandDetectPlan` in `index_pats_distributed()`, to be in line with the other two functions.
- Reversed the order of the pattern height and width in the `patDim` parameter passed to `EBSDIndex.update_file()`: the new order is (height, width).

4.3.3 Removed

- Parameter `filenameout` in functions `index_pats()` and `index_pats_distributed()`, as it is unused.

4.3.4 Fixed

- OpenCL kernels and test data are also included in the built distribution (wheel), not only the source distribution.

4.4 0.1.0 (2022-07-12)

4.4.1 Added

- Installation from Anaconda on Linux and Windows for Python 3.8 and 3.9.
- Make `ray` for parallel indexing an optional dependency, installable via the `pip` selector `pyebsdindex[parallel]`.
- Add `pip` selector `pyebsdindex[all]` for installing both `ray` and `pyopencl` to get parallel and GPU supported indexing.
- Support for Python 3.10.
- `ebsd_index` functions return both the orientation data and band identification data from the Radon transform.
- QUEST algorithm to get a best fit for the orientation.
- Many small improvements to Radon peak detection.
- PC conventions for Bruker, EDAX, EMsoft, kikuchipy, and Oxford.

4.4.2 Fixed

- Minimum version of `ray` package set to >= 1.13.
- Maximum version of `ray` package set to < 1.12.0 to avoid an import error on Windows.

CHAPTER

FIVE

INSTALLATION

PyEBSDIndex can be installed with pip or conda:

pip

```
pip install pyebsdindex[all]
```

conda

```
conda install pyebsdindex -c conda-forge
```

Further details are available in the *installation guide*.

CHAPTER

SIX

LEARNING RESOURCES

Tutorials

In-depth guides for using PyEBSDIndex.

API reference

Descriptions of functions, modules, and objects in PyEBSDIndex.

CHAPTER

SEVEN

CONTRIBUTING

PyEBSDIndex is a community project maintained for and by its users. There are many ways you can help!

- Report a bug or request a feature on GitHub
- or improve the *documentation or code*

PYTHON MODULE INDEX

p

INDEX