JOURNAL OF
APPLIED
CRYSTALLOGRAPHY

Volume 57 (2024)

Supporting information for article:

Fast Radon transforms for high-precision EBSD orientation determination using *PyEBSDIndex*

David J. Rowenhorst, Patrick G. Callahan and Håkon W. Ånes

Supplement: Fast Radon transforms for high precision EBSD orientation determination using PyEBSDIndex

D.J. Rowenhorst[1] P.G. Callahan[1] H. W. Ånes[2]

[1] U.S. Naval Research Laboratory, Materials Science and Technology Division, Washington, DC, USA

[2] Norwegian University of Science and Technology, Department of Materials Science and Engineering, Trondheim, Norway

1 Data availability

Example data, and the ability to recreate the simulated pattern benchmark data are provided at DOI:10.5281/zenodo.8400425 and DOI:10.18126/H2VW-RRZU

This includes:

- An hdf5 file that contains as-simulated, non-background corrected EBSD patterns and their simulated orientations which were used to benchmark the accuracy of PyEBSDIndex.
- A Jupyter notebook that explains how to create the patterns using the described noise model, index the patterns, and compare to the simulated orientations.
- A sample dataset of Ti-6Al-4V EBSD patterns in an EDAX .up1 file format.
- Jupyter notebook that demonstrates how to use NLPAR and PyEBSDIndex to index the patterns, and reproduce the data presented in the main article.

2 Sub-pixel peak localization

As mentioned in the main article, the location of the Kikuchi band peaks in the Radon transform can be calculated at a precision higher than integer location of the pixel values. This is done by fitting a quadratic equation to the pixel intensities, $\boldsymbol{A}$ of the peak location at (x_0, y_0) and its nearest-neighbor pixels

$$\boldsymbol{A} = \begin{bmatrix} I(x_0-1, y_0+1) & I(x_0, y_0+1) & I(x_0+1, y_0+1)) \\ I(x_0-1, y_0) & I(x_0, y_0) & I(x_0+1, y_0)) \\ I(x_0-1, y_0-1) & I(x_0, y_0-1) & I(x_0+1, y_0-1)) \end{bmatrix} \tag{1}$$

The intensity of this neighborhood of pixels is approximated as a 2-D second order Taylor series at any location, (x, y):

$$\begin{aligned} I\left[x,y\right] \approx & I[x_0,y_0] + \partial x[x_0,y_0](x-x_0) + \partial y[x_0,y_0](y-y_0) + \\ & \frac{\partial x^2[x_0,y_0]}{2}(x-x_0)^2 + \partial xy[x_0,y_0](x-x_0)(y-y_0) + \frac{\partial y^2[x_0,y_0]}{2}(y-y_0)^2 \end{aligned} \tag{2}$$

where $\partial x[x_0, y_0]$ is the partial derivative of the intensity with respect to x and $\partial x^2[x_0, y_0]$ is the second partial derivative evaluated at the location $[x_0, y_0]$. Here, these are approximated with finite difference kernels operating on the nearest-neighbor intensity matrix, $\boldsymbol{A}$:

$$\begin{aligned} \partial x \approx \Delta_x = \frac{1}{2}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \star \boldsymbol{A} \qquad \partial y \approx \Delta_y = \frac{1}{2}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \star \boldsymbol{A} \\ \partial x^2 \approx \Delta_{xx} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} \star \boldsymbol{A} \qquad \partial y^2 \approx \Delta_{yy} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} \star \boldsymbol{A} \\ \partial x \partial y \approx \Delta_{xy} = \frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \star \boldsymbol{A} \end{aligned} \tag{3}$$

where $\star$ indicates the matrix cross-correlation operator[1]. To solve for the location of the maximum relative to (x_0, y_0), the gradient of the Taylor quadratic is set to zero, providing two equations with two unknowns; substituting in $\delta x = x - x_0$ and $\delta y = y - y_0$:

$$0 = \nabla I(x,y) = \begin{bmatrix} \partial_x I(x,y) \\ \partial_y I(x,y) \end{bmatrix} = \begin{bmatrix} \Delta_x + \Delta_{xx}\delta x + \Delta_{xy}\delta y \\ \Delta_y + \Delta_{yy}\delta y + \Delta_{xy}\delta x \end{bmatrix} \tag{4}$$

Which is rearranged to:

$$\begin{bmatrix} -\Delta_x \\ -\Delta_y \end{bmatrix} = \begin{bmatrix} \Delta_{xx} & \Delta_{xy} \\ \Delta_{xy} & \Delta_{yy} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix} \tag{5}$$

This system of equations can then be solved using Cramer's rule, where the determinant is defined as:

$$D = \Delta_{xx}\Delta_{yy} - (\Delta_{xy})^2 \tag{6}$$

and thus:

$$\begin{aligned} \delta x = (\Delta_{xy}\Delta_y - \Delta_{yy}\Delta_x)/D \\ \delta y = (\Delta_{xy}\Delta_x - \Delta_{xx}\Delta_y)/D \end{aligned}$$

thus providing the sub-pixel location of the peak maxima (x, y). Literature has shown that this can effectively interpolate the sub-pixel location to about 0.1 pixel width [1].

[1]There can be some level of confusion about convolution and correlation within image processing. Formally, in a 2-D convolution the kernel is flipped horizontally, then vertically before the element-wise multiplication and summation, while with cross-correlation, there is no flipping operations. Since most of the kernels of interest in image processing (and their subsequent incorporation in convolutional neural nets) are symmetric, the two operations provide the identical result. Here, it is sufficient to say that because the local intensity matrix and the finite difference matrices are the same size, the cross-correlation is equivalent to an element-wise multiplication between the two matrices, and then summing all the rows and columns of the result.

3 Statistics from simulated pattern test

The attached table provides further statistics for the indexing accuracy test with simulated patterns. As stated in the main paper, for all methods, the known pattern center ($[0.5, 0.7, 0.6]$ using the EDAX pattern center notation) was used.

There were two additional indexing trials presented here for PyEBSDIndex using different pattern dimensions, a set of 60×60 px patterns, and 240×240 px patterns. A number of Radon dimensions with corresponding, DoG kernel sizes were tried for both sets. Interestingly, for the 60×60 px, using Radon dimensions of $N_\theta, N_\rho = [180, 120]$ and $\sigma_\theta, \sigma_\rho = [2.0, 2.9]$ provided only marginally worse accuracy as when these same Radon parameters with the 120×120 px patterns. Conversely, there was no significant change in accuracy when indexing the 240×240 px patterns using Radon dimensions of $N_\theta, N_\rho = [360, 240]$ and $\sigma_\theta, \sigma_\rho = [4.0, 7.8]$. The reasons for this are somewhat uncertain, other than the noted known inaccuracies in PyEBSDIndex of not accounting for all the distortions associated with the gnomonic projection, and using a fixed kernel size. An alternative explanation is that the 120×120 patterns are near the resolution limits of the master pattern used to create the simulated patterns. However, this seems highly unlikely as a rough estimate of the subtended angle of the detector at the simulated pattern center indicates that a full resolution projected pattern would contain about 1000 pixels per side from the 2k$\times$2k master pattern.

Indexing Method	Pattern Size (pixels)	PSNR (dB)	Min Error (deg.)	Mean Error (deg.)	95 % Confidence (deg.)	Max Error (deg.)	% Indexed Correctly ($< 2°$)	% No solution
PyEBSDIndex	60×60 $(N_\theta, N_\rho = (90, 60))$	AS	0.001	0.088	0.178	1.12	100	0
PyEBSDIndex	60×60 $(N_\theta, N_\rho = (180, 120))$	AS	0.001	0.055	0.109	0.55	100	0
PyEBSDIndex	240×240 $(N_\theta, N_\rho = (180, 120))$	AS	0.000	0.033	0.065	0.52	100	0
PyEBSDIndex	240×240 $(N_\theta, N_\rho = (360, 240))$	AS	0.000	0.034	0.068	0.29	100	0
PyEBSDIndex	120×120	AS	0.001	0.037	0.073	0.45	100	0
EMDI & BOBYQA	120×120	AS	0.000	0.014	0.030	0.14	100	0
EMSphInx BW 113	120×120	AS	0.001	0.119	0.178	1.53	100	0
EMSphInx BW 63	120×120	AS	0.005	0.207	0.310	2.00	99.97	0
Vendor Hough	120×120	AS	0.019	0.441	0.636	2.00	99.35	0
PyEBSDIndex	120×120	25	0.001	0.086	0.175	0.73	100	0
EMDI & BOBYQA	120×120	25	0.000	0.034	0.077	0.28	100	0
EMSphInx BW 113	120×120	25	0.001	0.085	0.144	1.39	100	0
EMSphInx BW 63	120×120	25	0.002	0.151	0.279	2.00	99.98	0
Vendor Hough	120×120	25	0.012	0.451	0.714	2.00	99.31	0
PyEBSDIndex	120×120	20	0.001	0.152	0.315	1.28	100	0
EMDI & BOBYQA	120×120	20	0.000	0.054	0.124	0.81	100	0
EMSphInx BW 113	120×120	20	0.002	0.112	0.206	1.57	100	0
EMSphInx BW 63	120×120	20	0.002	0.203	0.400	2.00	99.98	0
Vendor Hough	120×120	20	0.010	0.480	0.799	2.00	99.27	0
PyEBSDIndex	120×120	12	0.003	0.536	51.004	2.00	84.26	1.135
EMDI & BOBYQA	120×120	12	0.000	0.187	0.528	1.99	99.96	0
EMSphInx BW 113	120×120	12	0.003	0.356	0.765	2.00	99.99	0
EMSphInx BW 63	120×120	12	0.007	0.626	1.408	2.00	99.05	0
Vendor Hough	120×120	12	0.008	0.780	55.021	2.00	69.18	0

Table 1: Results of simulated pattern indexing error. AS indicates "as-simulated" patterns with no noise, which is equivalent to an infinite PSNR value. Unless otherwise noted, the Radon dimensions used were $N_\theta, N_\rho = [180, 120]$

4 Notes on optimizing GPU compute for EBSD patterns

Graphic processing units (GPUs), as the name implies, are exceptionally well suited for manipulating images, using thousands of threads to operate in parallel. However, the analysis of millions of EBSD patterns presents an interesting case for GPU compute. The typical situation envisioned for most image processing pipelines is a set of filters designed to operate on relatively large photo-like images, with dimensions of 10s –100s of megapixels not unusual. Thus, most of the examples for this type of work concentrate on breaking the image into several different tiles, and understanding how each thread and workgroup can most efficiently work on each tile of the larger image. EBSD patterns can reach the lower end of these image dimensions, but these cases are relatively rare, and it is much more common to work with large batches of images that are (60–240 px)^2, thus making it impractical to break them into tiles.

One common bottleneck for GPU compute is the time penalty for fetching data from the global memory of the GPU into the kernel memory of the operating thread. Again, if operating on a single large image, one strategy is to operate on vectors of values that are located in consecutive memory locations on the global memory block, which allows a single vector call to global memory to fetch multiple values at once. PyEBSDIndex has altered this strategy for working with batches of small images. The typical method to load a series of patterns into computer memory is to read the pattern in with column values incrementing fastest, then rows, then slices (in this case slice is each individual pattern), and indeed this is how all EBSD patterns the authors have encountered have been stored inside of files. PyEBSDIndex will read in a batch of patterns, and then transpose them in memory so that the slice direction increments fastest within memory, then the columns, then the rows. This then allows for using long vectors to batch process the image pipeline. For instance, a call to perform a calculation on the first column, first row would request a `float16`, indicating fetching a 16 element vector from memory at the specified location. Assuming that the batch of patterns has at least 16 patterns, this then would grab the first column-row for the first 16 patterns, and perform the same operation on all 16 values. Naturally, if the batch of patterns is not an even factor of 16, then PyEBSDIndex will automatically pad out the needed extra patterns. While there is overhead for performing the re-ordering of the data in memory, this is significantly overcome by the dramatic reduction of the number of fetches to global memory needed during the image processing on the GPU. And it should be also noted that the Radon and convolutional arrays created on the GPUs are laid out in a similar manner, with the pattern slice value incrementing fastest in memory. In the authors experience, this change lead to an approximately 3–5$\times$ increase in computational speed on the GPU compared to not using the vectorized memory fetches.

As indicated, PyEBSDIndex operates on all the patterns as using floating point math, with each pattern and Radon using 32-bit floats. The authors see no significant advantage to using the extra precision of 64-bit floats in these calculations, thus allowing for using the more common consumer/gaming GPUs for compute, rather than workstation grade GPUs which have dedicated computational units for 64-bit floats.

The authors found that one can extract more performance from a single GPU by running multiple calculation queues on one physical GPU. This is likely because if a single queue is used, then it is sitting idle while the indexing occurs on the CPU. Furthermore, the GPU can schedule simultaneous tasks such as having one queue transfer data from the main system memory to the GPU, at the same time that the another queue is calculating. We suspect that this is all very particular to the exact nature of the calculations within PyEBSDIndex, the authors find that 8–12 simultaneous queues are optimal. PyEBSDIndex will automatically schedule the GPU queues, as well as decide on a reasonable batch size for the amount of GPU memory, thus removing the need for the user to tune these parameters. It is unclear to what degree this is particular to OpenCL, or even each platform's implementation of OpenCL.

Finally, the authors admit that it is likely the solutions presented here could be far from optimal, and that there are many computational efficiencies that are not being leveraged. However, we present this information for others that may want to engage in similar pursuits, and leave it to the community to decide in the future if it is a guide for what-to-do, or what-not-to-do.

References

[1] Alfonso Alba et al. "Phase correlation with sub-pixel accuracy: A comparative study in 1D and 2D". In: *Computer Vision and Image Understanding* 137 (2015), pp. 76–87. DOI: 10.1016/j.cviu.2015.03.011.