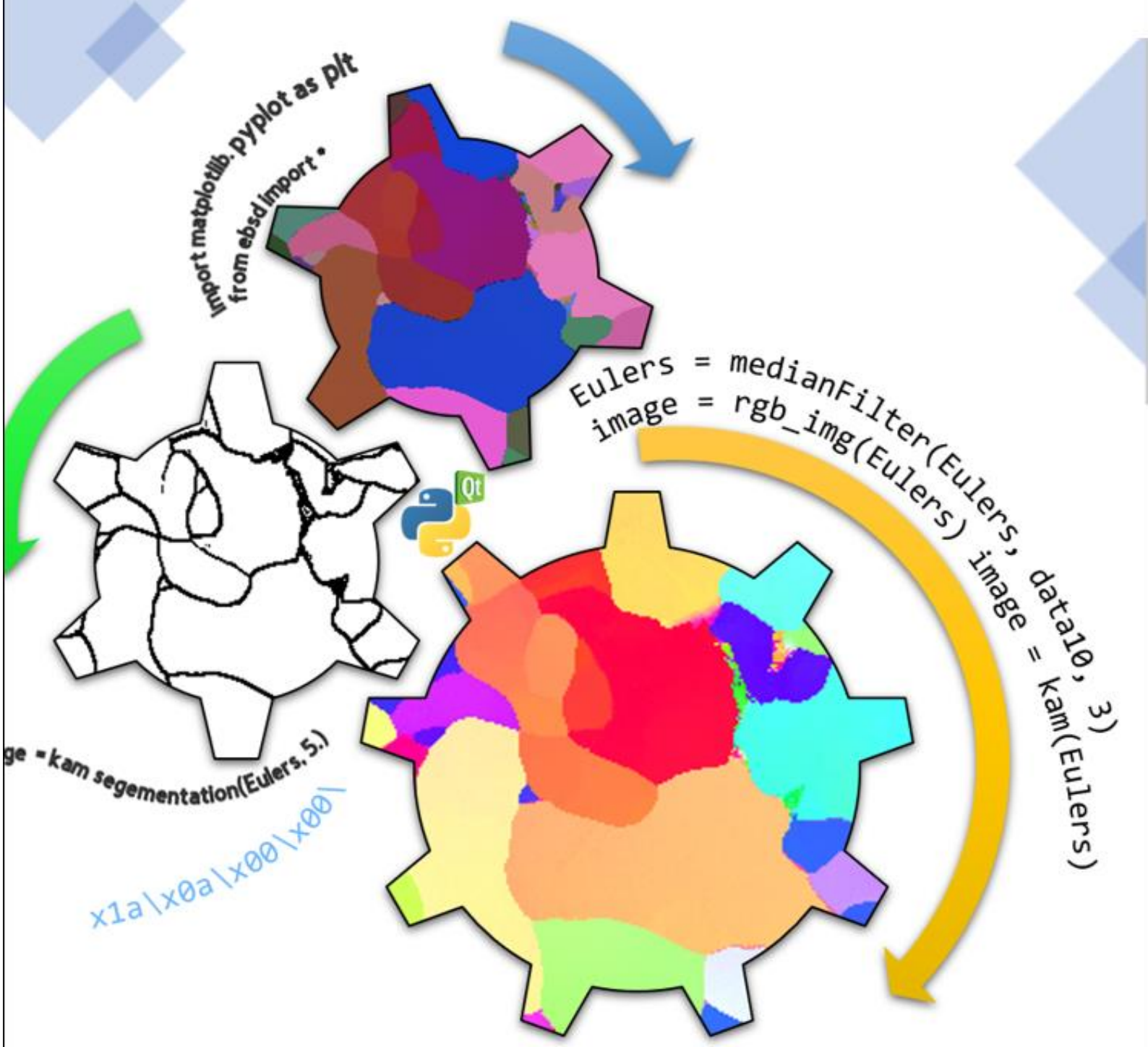


Grain Segmentation and Analysis of EBSD Images

by Harini T

BONAFIDE CERTIFICATE

This is to certify that the project titled "**GRAIN SEGMENTATION AND ANALYSIS OF EBSD IMAGES**" is a bonafide record of the work done by

HARINI. T (2020BCS0095)

as part of her B.Tech (First year) project work from **INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, KOTTAYAM (IIITK),** under my guidance during April 2021 – June 2021.

Dr. Anish Kumar
Head, Non Destructive Evaluation Division
Professor and Dean-Academic (Engineering Sciences)
Homi Bhabha National Institute
Indira Gandhi Centre for Atomic Research
Kalpakkam-603 102, Tamil Nadu, India

ACKNOWLEDGEMENT

I am thankful to the management of Indira Gandhi Centre for Atomic Research (IGCAR) for permitting to carry out this project in this esteemed institute. I thank **Dr. Ebin Deni Raj**, HOD (Computer Science & Engineering) IIIT-K and Dr. **Rajiv V. Dharaskar**, Director, IIIT Kottayam for permitting me to carry out this project.

I express my sincere gratitude to my guide **Dr. Anish Kumar**, Head, Nondestructive Evaluation Division and Dean, Engineering Sciences, HBNI, (IGCAR), Kalpakkam, for offering this project and providing constant guidance that led to the successful completion of this work. I like to place on record that the knowledge and research prowess I gained from him during the course of this work will certainly benefit me in my career.

I express my deep sense of gratitude to **Dr. Vani Shankar** for the encouragement and support provided during the course of this project and providing the required material microstructure images.

I am thankful to all the Scientists and non-technical staff in the Nondestructive Evaluation Division, IGCAR for their welcoming and accommodating attitude during my visits to IGCAR.

I wish to express gratitude to my family for the patience, understanding and constant support throughout the project duration.

Contents

Chapter 1. Introduction

1.1. Introduction to Electron Back Scattered Diffraction (EBSD)

Electron backscatter diffraction (EBSD) is a technique that can determine the local crystal structure and crystal orientation at the surface of a specimen [1]. The methodology collects elastically back scattered electrons (BSEs) which have undergone coherent Bragg scattering as they leave the specimen. A dedicated EBSD detector collects the scattered BSEs over a large solid angle, which forms electron backscatter diffraction patterns made up of Kikuchi bands as schematically shown in Figure 1. The EBSD patterns can be analyzed to give the crystalline structure and orientation of the crystal. EBSD analysis in SEM is now very automated and has found widespread application in the analysis of crystallography of metallic alloys.

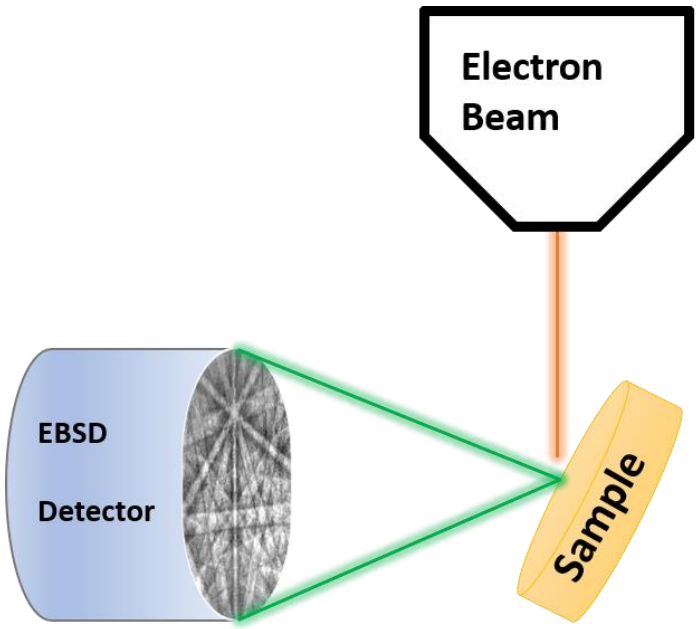


Figure 1 Schematic representation of an EBSD measurement setup

The orientation between specimen coordinate system and the EBSD system can be defined by a set of three successive rotations about specified axes. These rotations are called the Euler angles and are shown in Figure 2. The Euler angles can be used to quantify which crystal structures are present and their orientation, sizes and morphology of individual grains, the collective texture of alloys, and crystallographic relationships between phases. However, this requires the development of computational algorithms and dedicated software frame work. This project is aimed to develop a dedicated software using Python development environment.

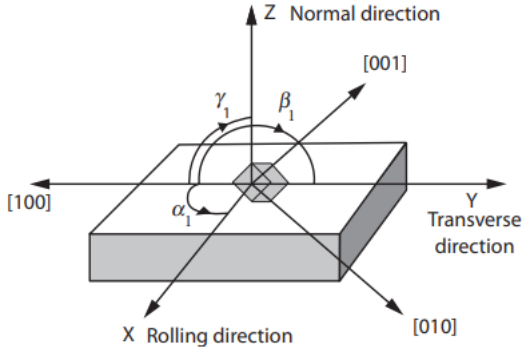


Figure 2 Relationship between the specimen coordinate system XYZ (or RD, TD, ND) and the crystal coordinate system 100,010,001 where the cubic unit cell of one crystal in the specimen is depicted.

In this work the Python programming language has been chosen for its versatility, rich collection of library functions and ease of coding. In addition to the development of algorithms using Python programming, a graphical user interface design and implementation has also been carried out for user friendliness using PyQt development tools. PyQt is a Python binding of the cross-platform GUI toolkit Qt, implemented as a Python plug-in.

1.2. EBSD measurement data and rendering RGB image

In a typical EBSD measurement, the three Euler angles are stored in a binary file with header information. The first and foremost job is to read this data file and create a Euler image. In the Euler angles the angle 1 represent the red colour, Euler angle 2 represent the green color and Euler angle 3 represent blue colour. The header contains information such as scan pitch in X and Y directions, number of scan points (image size) etc., A python program has been developed to read the Euler angle file and render an RGB image. In this case the Euler angle 1 can have values from 0 to 360 while and other two angles vary from 0 to 90 degrees. Normalisation has been done to push the range from 0-255 for representing them in RGB images. The python code snippet for normalisation of each angle to represent proportionate color value is given below:

```
rgb[:,:,i] = (255/Max.Value)*ei
rgb = np.uint8(rgb)
#where i = 0,1,2
```

Typical pseudo colour Euler angle images and the RGB Euler image of a ferrite microstructure is shown in Figure 3.

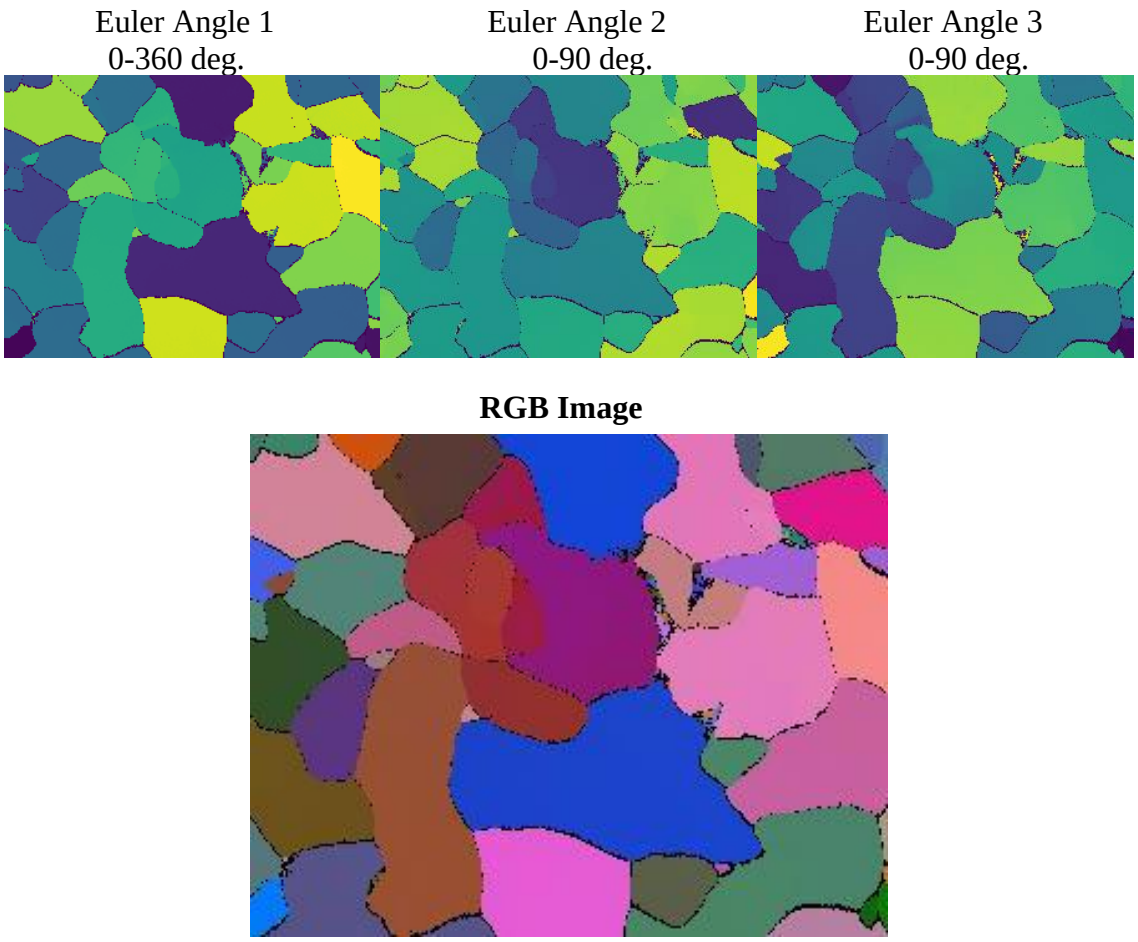


Figure 3 Individual pseudo colour Euler angle images and the RGB image combining all the three Euler angles.

A schematic flow chart of the software proposed to be developed is shown in Figure 4. As can be observed, the software will accept the Euler angle data of the EBSD measurements on a given specimen. The data will be read to render an RGB image as outlined earlier. The RGB image will be subjected pre-processing to eliminate salt and pepper type of noise due to scattering of electrons at the grain boundary and missing data points. The pre-processed image will further be subjected to segmentation operation to extract grain properties such as grain boundary, region properties, kernel average misorientation, to generate maps such as pole figure, inverse pole figure, dislocation density and grain reference orientation deviation.

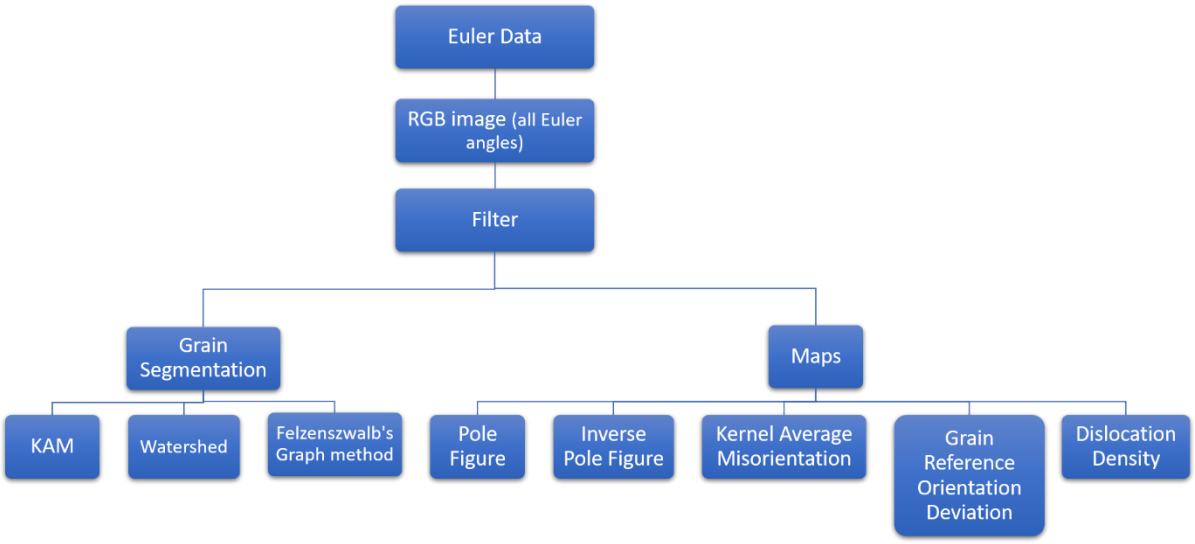


Figure 4 Flow chart of the software development for analysis of EBSD data

1.3. Filtering of the RGB images

As mentioned earlier the raw Euler images contain many zeros due to scattering of electrons at the grain boundaries which would not reach the detector. This will introduce salt and pepper noise in the EBSD images. These images are subjected to median filtering to smooth the data. Median filtering is a sliding window technique using 3x3 or 5x5 or higher kernel matrices based on the median of the data in the image superposing the kernel. Typical sliding operation is depicted in Figure 5. After the median filtering the zeros are replaced by the median number within the kernel. The raw RGB Euler image before and after median filtering is shown in Figure

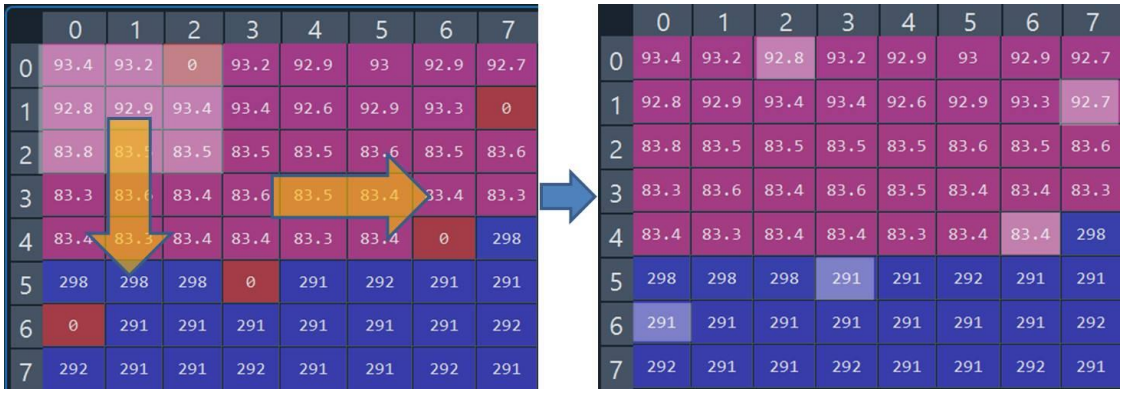
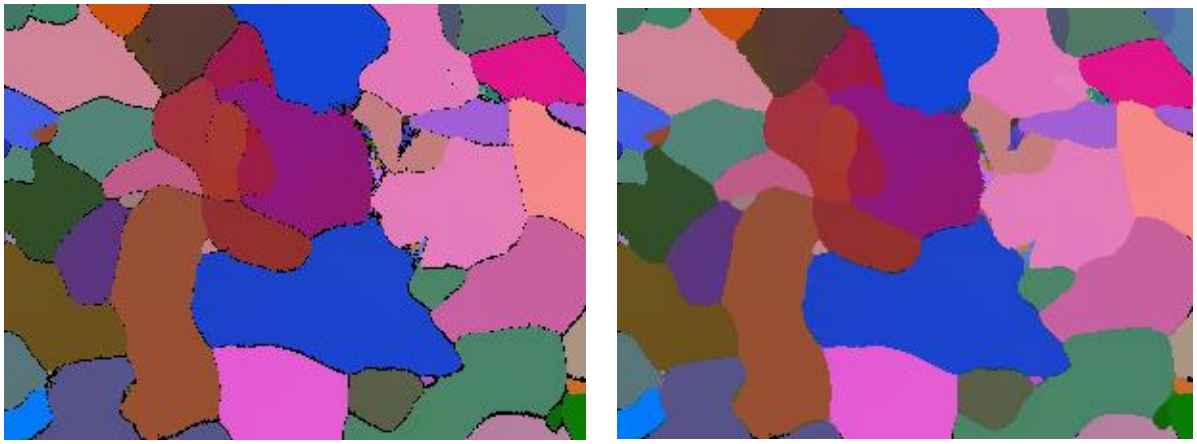


Figure 5 Schematic representation of the sliding window operation during the median filtering of the image data.



a) b)

Figure 6 Raw and b) Median filtered RGB Euler images.

Next section deals with the segmentation operation on the filtered RGB images to extract different features mentioned above and the orientation maps in the further sections.

Chapter 2. Segmentation of EBSD Images

One of the main goals of EBSD image analysis is the grain segmentation. It is essential for an user to understand the properties and orientations of the grains present in the EBSD images to enable material characterization and sample manipulation for a range of applications.

Grain segmentation in this project has been achieved with the following approaches:

- Standard image segmentation algorithms and
- Kernel Average Misorientation based segmentation

2.1. Standard image segmentation techniques

The standard image segmentation algorithms are broadly classified into edge-based, region-based and clustering based methods.

Edge-based segmentation involves the separation of foreground from the background of an image, which in the case of a Euler image will be only to detect the edges of the grain (background) using a threshold value based on Otsu's method [2]. This segmentation is not helpful in extracting individual grain as regions.

Region-based segmentation encompass, pre-processing of the images that includes edge detection and applying distance transformation, after which watershed algorithm on the image is performed for segmentation.

K-means, a popular clustering method for image segmentation groups a number of grains into a single cluster. For K-means to segment each grain separately we must know the total number of grains beforehand so that the number of clusters can be equal to the number of grains. So, this method was not feasible in this case.

In this study region-based segmentation technique has been chosen due to its versatility and ease of extracting features of the individual grains in an Euler image [3]. This essentially involves following three major steps: i) Edge detection, ii) Distance transform and iii) Watershed.

Edges in an image (I) are detected by convolving the Sobel operator G_x in the horizontal direction and G_y in the vertical direction, where

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \text{ and } G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Gradient Magnitude, $G = \sqrt{{G_x}^2 + {G_y}^2}$

The edge detected image was converted to binary using a threshold value (threshold Otsu function). The binary image is multiplied with the original image to obtain an image with clearly defined edges.

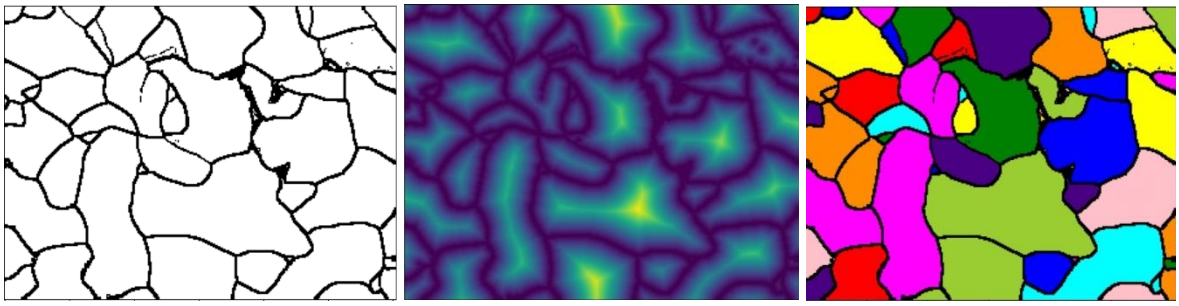
The Grain segmentation algorithm using watershed involves the following steps:

- Step 1: The edge detected binary image where the grains are in foreground and the rest as the background (edges) is used for further processing
- Step 2: Compute the Euclidean distance of the grain regions to the background using distance transform.
- Step 3: Compute the local maxima of the distance transform which would be the grain center.
- Step 4: Using the local maxima as markers and the distance transform as segmentation functions compute the watershed transform to segment the regions

On segmenting the grains, we could easily extract region properties such as mean Euler angles (grain orientation), centroids, aspect ratio, area etc. which help us understand and analyze the crystalline material.

Many other region-based segmentation techniques such as Felzenszwalb's (Graph based method), Simple Linear Iterative Clustering (SLIC) and Quickshift (Kernelized mean shift) have also been attempted to segment Euler image for grain segmentation of which Felzenszwalb's method produces similar results to the watershed technique. The above mentioned techniques include many free parameters such as maximum iteration, compactness, scale, minimum size of region. Hence, fixing the parameters in relation with the angles is tedious and cumbersome. Moreover, over-segmentation has also been observed in a few cases.

Python routines have been developed to carry out region-based segmentation of the Euler images. The code takes the Euler RGB image as input and provides the segmented image. It also provides the grain labels so that further grain level operations can be performed. The resulting images in the sequence of edge detection, distance transform and watershed are shown in Figure 7 for the ferrite EBSD image data shown in Section 1. As can be seen in the edge detected binary image, each grain boundary has been clearly extracted using the Sobel operator and Otsu threshold function. The distance transformed image clearly shows the local maxima regions of each grain. Finally, grains have been efficiently extracted and are shown in different color in the watershed image.



a) b) c)

Figure 7 Region based image segmentation results for the EBSD image of ferrite microstructure; a) Edge detected binary image, b) Distance transform image and c) Watershed image

2.2. Kernel Average Misorientation (KAM) based Segmentation

Kernel Average Misorientation (KAM) is the average misorientation angle of a given point with all its neighbours. Misorientation is the difference in crystallographic orientation between two crystallites and describes the relative orientation of two crystals with respect to each other.

The KAM for a pixel at $(i\,, j)$ is defined as:

$$KAM_{i,j} = \frac{1}{|N(i,j)|} \sum_{(k,l)\in N(i,j)} \omega(o_{i,j}, o_{k,l})$$

$|\,N\,(i,\,j)\,|$ - number of all neighbors taken for calculating misorientation angle

$\omega(o_{i,j}, o_{k,l})$ – Misorientation angle between the pixel $(i,\,j)$ and its neighbor $(k,\,l)$

The three Euler angels are obtained for any pixel $(i,\,j)$ from the Euler array with the respective indices. The rotation matrices (z-x-z) for the two pixels is calculated (say g_1 *and* g_2). The misorientation angle between any two pixel is calculated by the formula given below:

$$m \;=\; g_1^{-1} \times g_2 \;(g_1 and\; g_2\; are\; orthogonal\; matrices)$$

$$K \;=\; 0.5 \;\times\; [trace(m) \;-1\,]$$

$$\omega\big(o_{i,j}, o_{k,l}\big) = cos^{-1}(\,K) \times \left(\frac{180}{\pi}\right), degrees$$

The KAM based segmentation is a technique devised to segment the regions based the misorientation of the grain boundaries. Usually, four or eight-point neighbourhood is taken for calculating KAM. As the grain boundaries in the Euler image have large misorientation as compared to its neighborhood, it is natural choice for segmenting an image. Generally, the

misorientation of any two grains is observed to be greater than 15 degrees and the misorientation of Low-angle grain boundaries (LAGB) or sub grain boundaries are less than 15 degrees. In a typical Euler image of size 303 x 227, there are a total of 68781 pixels. As KAM is based on neighbourhood operation, it is in general a time consuming and a computationally intensive task. Hence, it is essential to develop optimized algorithms as well as improve the computational efficiency by way of parallelization. Python routines have also been developed for computing the KAM of a Euler image. The original RGB Euler and the KAM computed images using the developed Python code are shown in Figure 8.

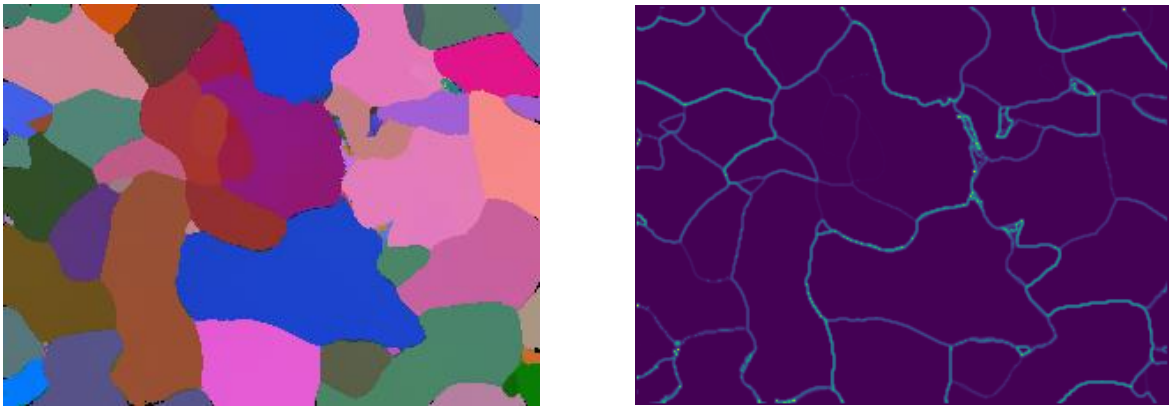


Figure 8 Original image and the computed Kernel Average Misorientation image using the Python code.

Code optimization has been performed based by simplifying a few matrix operations. In order to speed up the computation, Python based just in time (JIT) compiler using NUMBA and CUDA GPU coding have been explored. These studies have been presented in the next sections.

2.3. Optimization of KAM code for improving computation efficiency

The KAM code has been optimised in the following ways:

1) Rotation matrices (3) z-x-z has been changed into a single matrix reducing the number of matrix operations, which would result in less number of function calls.
2) The orthogonal property of rotation matrices (g_1, g_2) have been exploited to replace inverse function with transpose reducing computation time
3) Separate functions have been made for calculation of misorientation angle and the Kernel Average Misorientation for 8 neighbours
4) Usage of more NumPy array functions replacing of pythonic code to meet the compatibility with JIT compiler and further speed up.

With these modifications the run time reduced considerably, and further reduction of execution time is achieved using Numba JIT and GPU.

2.4. Numba Just-In-Time (JIT) Compiler

Numba offers an open-source JIT compiler that translates a subset of Python and NumPy into fast machine code using Low Level Virtual Machine (LLVM). It is specifically for numerical functions and allows you to accelerate your applications with high performance functions written directly in Python and also parallelize parts of a function. It provides many decorators/features such as **JIT** – just in time compiler (object mode) and **NJIT** – no object mode just in time compiler. Just in time (JIT) compilation is a way of executing computer code that involves compilation during execution of a program (at run time). A schematic representation of the function of the JIT compiler is shown in Figure 9.

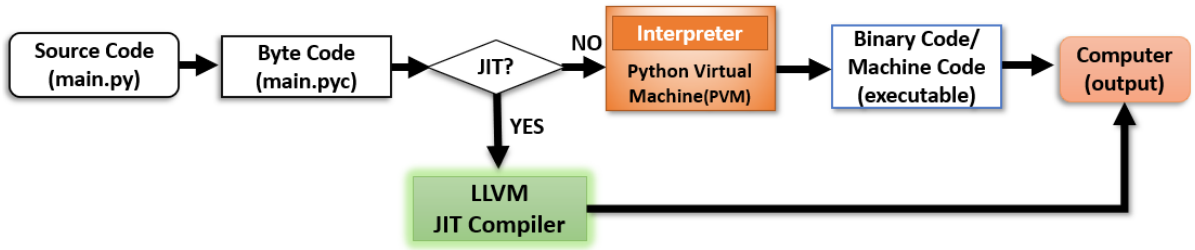


Figure 9 Schematic representation of the working of Just-in-Time compiler

The traditional way of executing a Python code is:

- The Python compiler reads a python source code or instruction and verifies that the instruction is well-formatted.
- If there is no error, then the compiler translates it into its equivalent form in an intermediate language called “Byte code”.
- Byte code is then sent to the Python Virtual Machine (PVM) which is the python interpreter. PVM converts the python byte code into machine-executable code.
- The Numba JIT compiler uses the Low-Level Virtual Machine (LLVM) compile library.
- As JIT allows a language to be interpreted and compiled during runtime, rather than at execution. It will be compiling the language as it is executing the logic prior to the code it is compiling thus, increasing the speed of execution.

2.5. Numba for CUDA GPUs

GPU programming has been explored in order to achieve further reduction in execution time. GPU is specialized for highly parallel computations. A CPU is designed to excel at executing a sequence of operation, called a thread as fast as possible and can execute a few tens of these threads in parallel, while the GPU is designed to excel at executing thousands of them in parallel. Numba offers a variety of options for parallelizing codes for CPUs and GPUs. Hence, GPU programming was attempted using Numba import library CUDA package. CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in **NVIDIA GPUs.** On the GPU, the computations are executed in separate blocks, and each of these blocks uses many threads. The set of all the blocks is called a grid. A schematic representation of a Grid, block and threads in a GPU is given in Figure 10. Each thread element is mapped to a data in the image matrix.

A block dimension of (16,16) has been taken based on the availability of GPU memory and the grid dimension in the program has been assigned as per the following formula:

$$griddim = \left(\frac{img.shape[0]}{blockdim[0]+1}, \frac{img.shape[1]}{blockdim[1]+1}\right)$$

The row and column values to access the elements in the image are given as per the formulae given below:

$$row = blockIdx.x * blockDim.x + threadIdx.x$$

$$col = blockIdx.y * blockDim.y + threadIdx.y$$

Each thread performs the function of calculating misorientation between 2 pixels $[(i,j)\ and\ one\ of\ its\ 8\ neighbors]$. And these threads are executed parallelly, independent of each other.

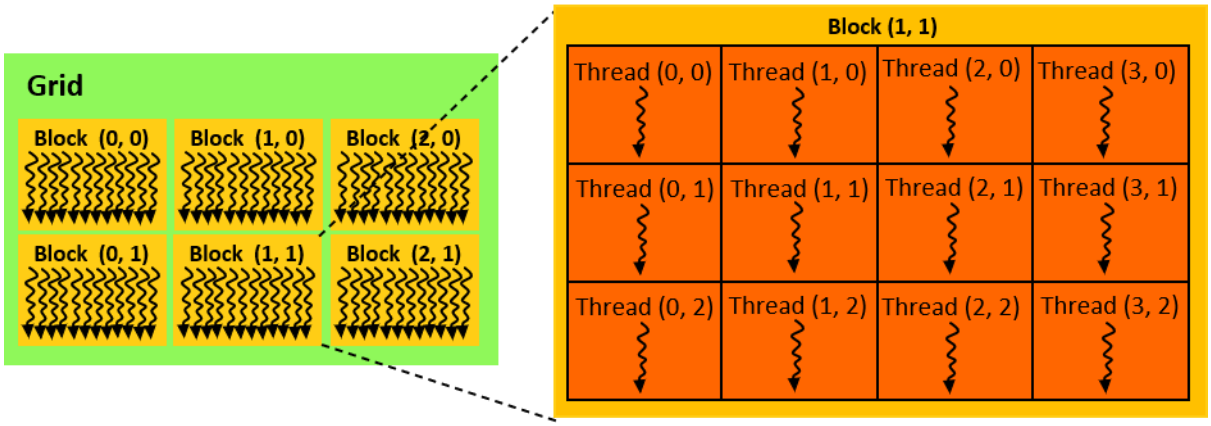


Figure 10 Schematic representation of a grid, blocks, and threads in a GPU.

The execution time, mean and standard deviation of execution times per loop for 7 times running with NJIT and GPU programming have been compared for EBSD images of size 1141 x1532 (Image-1), 334 x 740 (Image-2) and 227 x 300 (Image-3). The median filtered Euler RGB images of Image-1, Image-2 and Image-3 are shown in Figure 11.

Figure 11 RGB EBSD images of different sizes used for performance comparison of Python, NJIT and GPU codes

The computations have been performed in a standard laptop using Intel I3 Gen-7 processor, with 4 GB RAM and GPU memory of 4 GB running on windows 10 operating system. The computation time in normal Python code without any decorators, Numba NJIT and CUDA GPU codes are graphically shown in Figure 12. Further the execution time per loop for 7 times running the code for NJIT and GPU are summarized in Table 1. The execution time for NJIT and GPU are comparable. Nearly a 100 times reduction in the execution time is observed for the NJIT and GPU codes in comparison with the normal Python code for for all the images. With this improved performance of the KAM code, it is possible to carry out KAM based segmentation in a more efficient manner.

Table 1 Comparison of execution times per loop for 7 times running of the code for three images (158.332s =)

Program with	Image 1 (1141 x 1532)	Image 2 (334 x 740)	Image 3 (227 x 300)
Python	152000 ms ± 7840 ms	20500 ms ± 401 ms	8390 ms ± 409 ms
NUMBA NJIT	1370 ms ± 28 ms	189 ms ± 1.03 ms	50.6 ms ± 496 µs
GPU	1150 ms ± 11.1 ms	167 ms ± 1.13 ms	52.2ms ± 152 µs

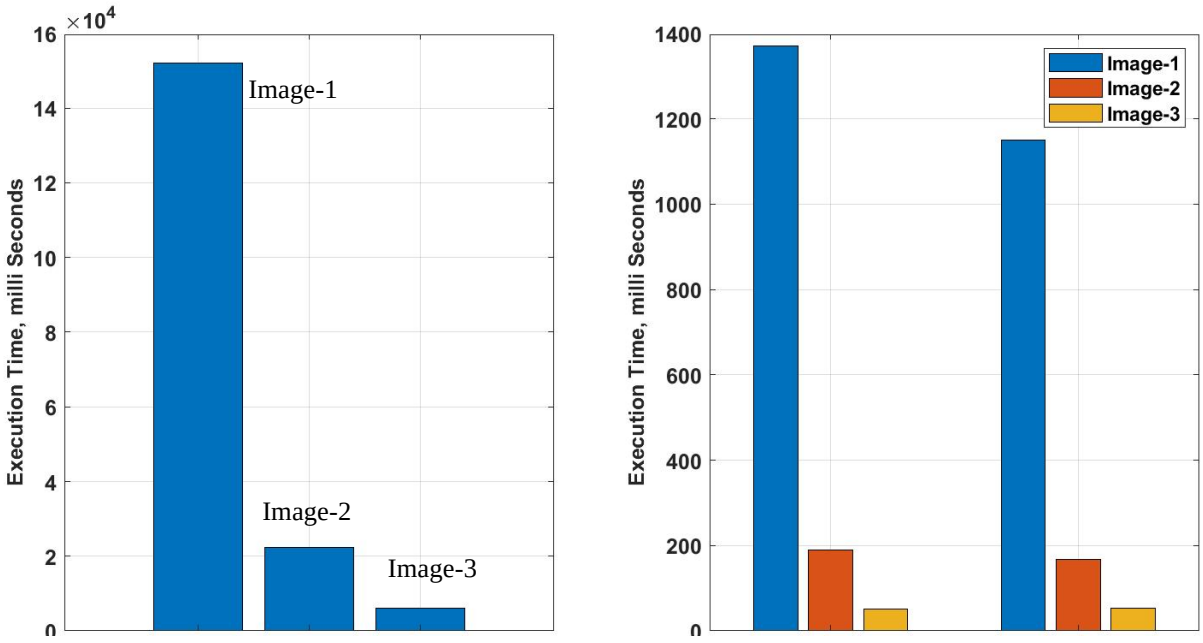


Figure 12 Execution times for the KAM program using Python, Numba NJIT and GPU.

2.6. KAM based Segmentation – Algorithm

Segmentation is carried out using an algorithm based on an orientation similarity criterion using stack data structure. The following are the sequence of steps for KAM based segmentation of the EBSD images:

- All pixels are initially marked as unlabeled using -1, and then pixels are evaluated one by one.
- A new grain is created, and non-assigned neighboring pixels are evaluated based on the crystal misorientation.
- If the misorientation is lower than `key`, the pixel is assigned to the current grain and its neighbors added to the list of points.

- When no more points are present, the next pixel is evaluated and a new grain is created.

Figure 13 shows the KAM based segmentation of a typical EBSD image. It is evident from the figure that the KAM based segmentation performs the best among all the segmentation methods as it facilitates segmentation based on misorientation angles and boundaries are detected with different intervals of angles.

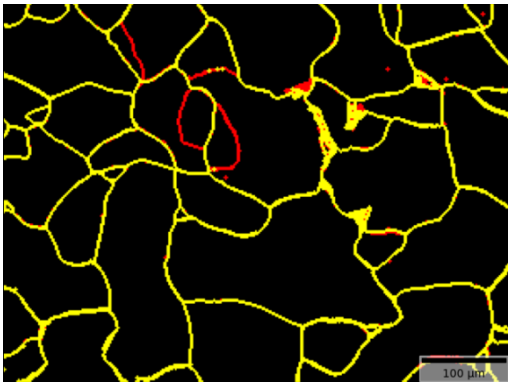
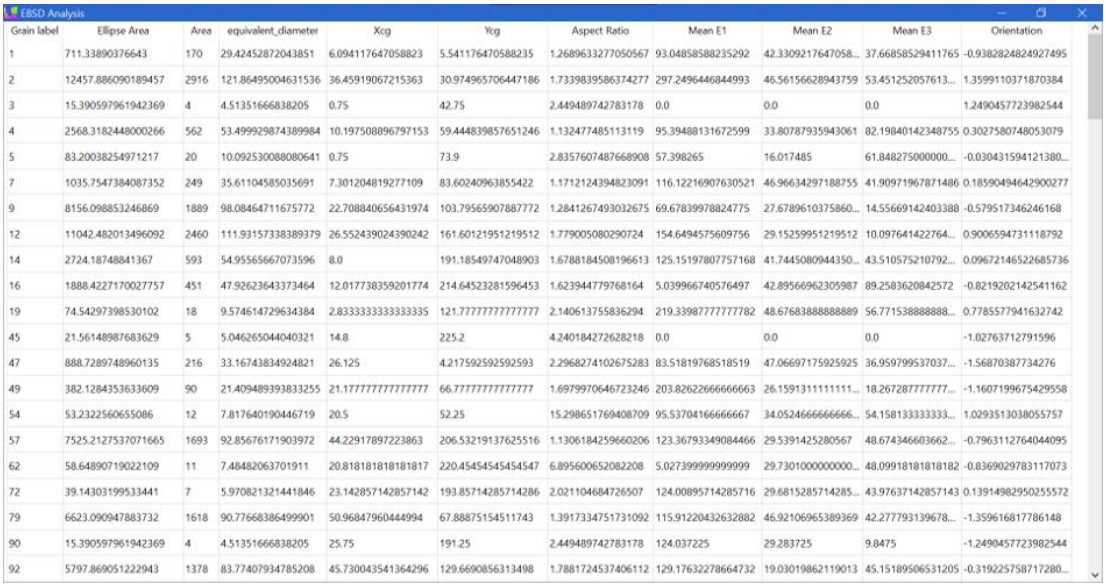


Figure 13 KAM based segmented image of ferrite microstructure. The grain boundary with misorientation angle >1° are given red colour, boundary with misorientation angle >5° are given yellow colour.

2.7. Extracting grain properties from segmented images

The segmented images enable us to label individual grains and extract different properties of the grains. A few properties can also be used to obtain various maps which are dealt in the next chapter. The grain properties include area, centre of gravity, aspect ratio, mean Euler angles and orientation. In addition, individual grain coordinates are extracted, and ellipse is fit to the grain boundary to compute the ellipse area and equivalent diameter of the grain. Python routines have been developed to compute for selection of a particular grain in an UI and provide the above mentioned properties or features of the grain. The grain features are also exported to an excel sheet with click of a mouse. Figure 14 shows the screen shot of the exported grain properties in the software.



EBSD Analysis

Grain label	Ellipse Area	Area	equivalent_diameter	Xcg	Ycg	Aspect Ratio	Mean E1	Mean E2	Mean E3	Orientation
1	711.33890376643	170	29.42452872043851	6.094117647058823	5.541176470588235	1.2689633277050567	93.04858588235292	42.3309217647058...	37.66858529411765	-0.9382824824927495
2	12457.886090189457	2916	121.86495004631536	36.45919067215363	30.974965706447186	1.7339839586374277	297.2496446844993	46.56156628943759	53.451252057613...	1.3599110371870384
3	15.390597961942369	4	4.51351666838205	0.75	42.75	2.449489742783178	0.0	0.0	0.0	1.2490457723982544
4	2568.3182448000266	562	53.499929874389984	10.197508896797153	59.444839857651246	1.132477485113119	95.39488131672599	33.80787935943061	82.19840142348755	0.3027580748053079
5	83.20038254971217	20	10.092530088080641	0.75	73.9	2.8357607487668908	57.398265	16.017485	61.848275000000...	-0.0304315941213880...
7	1035.7547384087352	249	35.61104585035691	7.301204819277109	83.60240963855422	1.1712124394823091	116.12216907630521	46.96634297188755	41.909719678714860	0.18590494642900277
9	8156.098853246869	1889	98.08464711675772	22.708840656431974	103.79565907887772	1.2841267493032675	69.67839978824775	27.67896103758600...	14.556691424033880	-0.5795173462461680
12	11042.482013496092	2460	111.93157338389379	26.552439024390242	161.60121951219512	1.779005080290724	154.6494575609756	29.15259951219512	10.097641422764...	0.9006594731118792
14	2724.18748841367	593	54.95565667073596	8.0	191.18549747048903	1.6788184508196613	125.15197807757168	41.7445080944350...	43.510575210792...	0.0967214652268573 6
16	1888.4227170027757	451	47.92623643373464	12.017738359201774	214.64523281596453	1.623944779768164	5.039966740576497	42.895669623059870	89.258362084257 2	-0.8219202142541162
19	74.54297398530102	18	9.574614729634384	2.8333333333333335	121.77777777777777	2.140613755836294	219.33987777777782	48.676838888888889	56.771538888888...	0.7785577941632742
45	21.56148987683629	5	5.046265044040321	14.8	225.2	4.240184272628218	0.0	0.0	0.0	-1.027637127915 96
47	888.7289748960135	216	33.16743834924821	26.125	4.217592592592593	2.2968274102675283	83.51819768518519	47.066971759259 25	36.959799537037...	-1.5687038773427 6
49	382.1284353633609	90	21.409489393833255	21.177777777777777	66.77777777777777	1.6979970646723246	203.82622666666663	26.159131111111...	18.267287777777...	-1.1607199675429558
54	53.2322560655086	12	7.817640190446719	20.5	52.25	15.298651769408709	95.53704166666667	34.052466666666...	54.158133333333...	1.0293513038055757
57	7525.2127537071665	1693	92.85676171903972	44.22917897223863	206.53219137625516	1.1306184259660206	123.36793349084466	29.539142528056 7	48.674346603662...	-0.7963112764044095
62	58.64890719022109	11	7.484820637019 11	20.818181818181817	220.45454545454547	6.895600652082208	5.027399999999999	29.730100000000...	48.099181818181 82	-0.8369029783117073
72	39.14303199533441	7	5.970821321441846	23.142857142857142	193.85714285714286	2.021104684726507	124.00895714285716	29.681528571428 5...	43.976371428571 43	0.1391498295025557 2
79	6623.090947883732	1618	90.77668386499901	50.96847960444994	67.88875154511743	1.3917334751731092	115.91220432632882	46.921069653893 69	42.277793139678...	-1.3596168177861 48
90	15.390597961942369	4	4.51351666838205	25.75	191.25	2.449489742783178	124.037225	29.283725	9.8475	-1.2490457723982544
92	5797.869051222943	1378	83.77407934785208	45.730043541364296	129.6690856313498	1.7881724537406112	129.17632278664732	19.030198621190 13	45.151895065312 05	-0.3192257587172 80...

Figure 14 Screen shot of exported grain properties computed using Python code to an excel sheet.

Chapter 3. Generating EBSD MAPS

The segmented images have also been used to extract maps such as pole figure and inverse pole figure [4,5,6]. In addition, it is also possible to estimate the dislocation density in a grain and compute grain reference orientation deviation. Python programs have been developed to accomplish this goal.

3.1. Pole Figure

Pole figure maps are a representation of 3-D orientation information of the crystal lattice onto the unit circle in two-dimension. Pole figure in the form of stereographic projections is used to represent the orientation distribution of crystallographic lattice planes in crystallography and texture analysis in materials science.

3.2. Stereographic Projection of Crystal Faces

Spherical projection is used to represent the crystallographic angles in a systematic way. In this projection, the crystal is assumed to be inside a sphere and the normal to the crystal faces that intersect the sphere is considered as the *poles* to the crystal face (Figure 15). The poles on the upper sphere are projected to the equatorial plane (XY plane) with respect to the pole (in this case $00\bar{1}$ shown in Figure 15) using the stereographic projection.

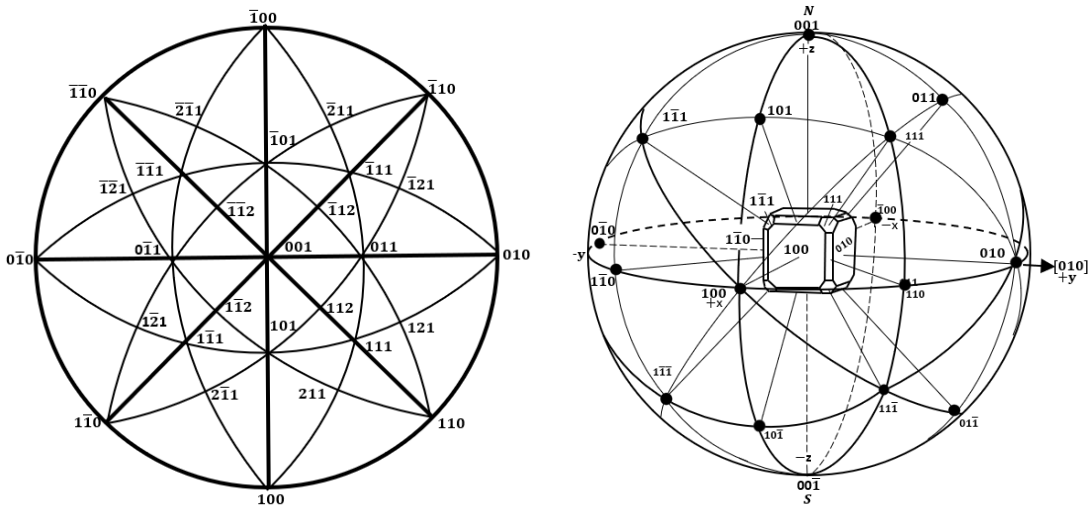


Figure 15 Poles of crystal faces (left). Stereographic projection of the poles on the equatorial plane – xy with respect to pole $00\bar{1}$ (right).

The stereographically projected pole points can be obtained as described below:

Equation of the line joining $(0, 0, -1)$ and (x_1, y_1, z_1):

$$\frac{x}{x_1} = \frac{y}{y_1} = \frac{z+1}{z_1+1}$$

The required point is the intersection point of this line and the XY- plane i.$e.$ $(z = 0)$

$$\Rightarrow x = \frac{x1}{1+z1} \text{ and } y = \frac{y1}{1+z1}$$

Similarly for projection {100} and {010} the formulae used are:

$$\frac{x+1}{x1+1} = \frac{y}{y1} = \frac{z}{z1} \quad (x = 0) \Rightarrow y = \frac{y1}{1+x1} \text{ and } z = \frac{z1}{1+x1}$$

$$\frac{x}{x1} = \frac{y+1}{y1+1} = \frac{z}{z1} \quad (y = 0) \Rightarrow x = \frac{x1}{1+y1} \text{ and } z = \frac{z1}{1+y1}$$

In order to make plotting of the stereographic projection easier to read, a device called a stereographic net or stereonets are used. Usually, one pole of a plane is not sufficient to represent the orientation of a grain in the pole figure map. So, three planes from a cube rotated the orientation of the grain is considered for the representation of a grain in the pole figure.

The GUI includes 3 different pole figure views as mentioned below:

1) Pole figure for a grain selected by the user (Figure 16)
2) Pole figure for a selection region in the segmented image (Figure 17)
3) Pole figure for all grain in an EBSD image (Figure 18)

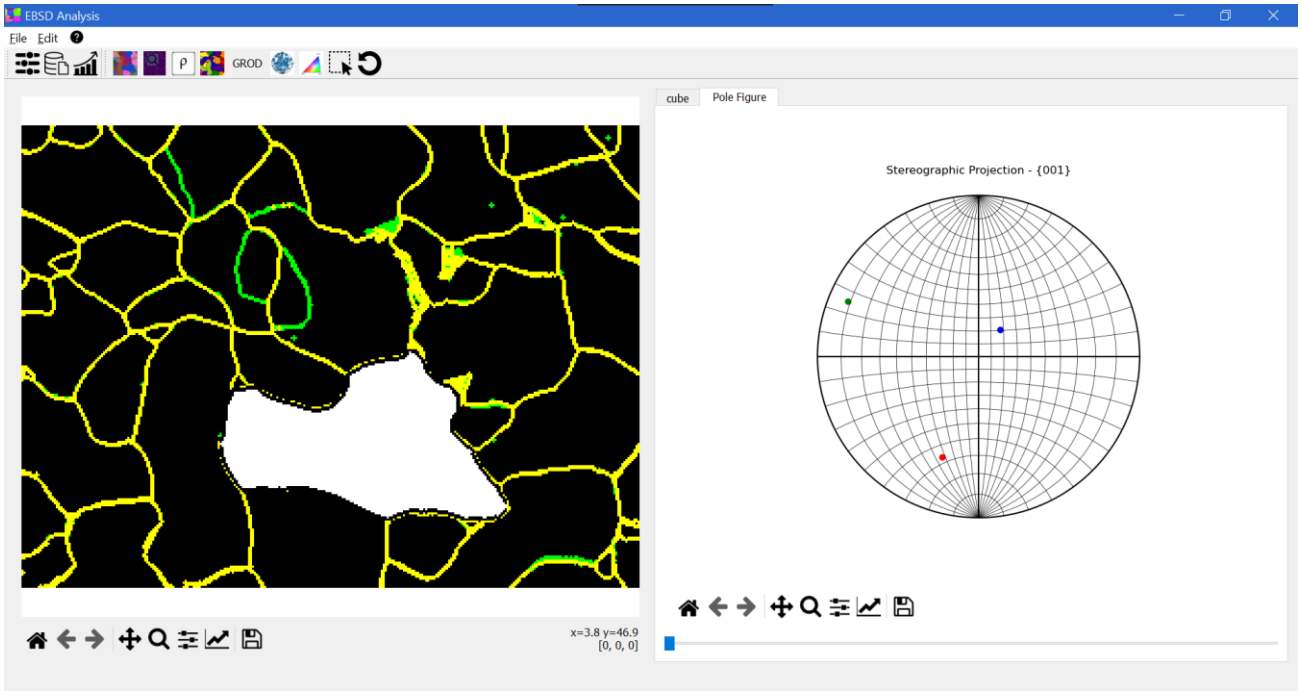


Figure 16 Pole figure for a grain selected by the user,

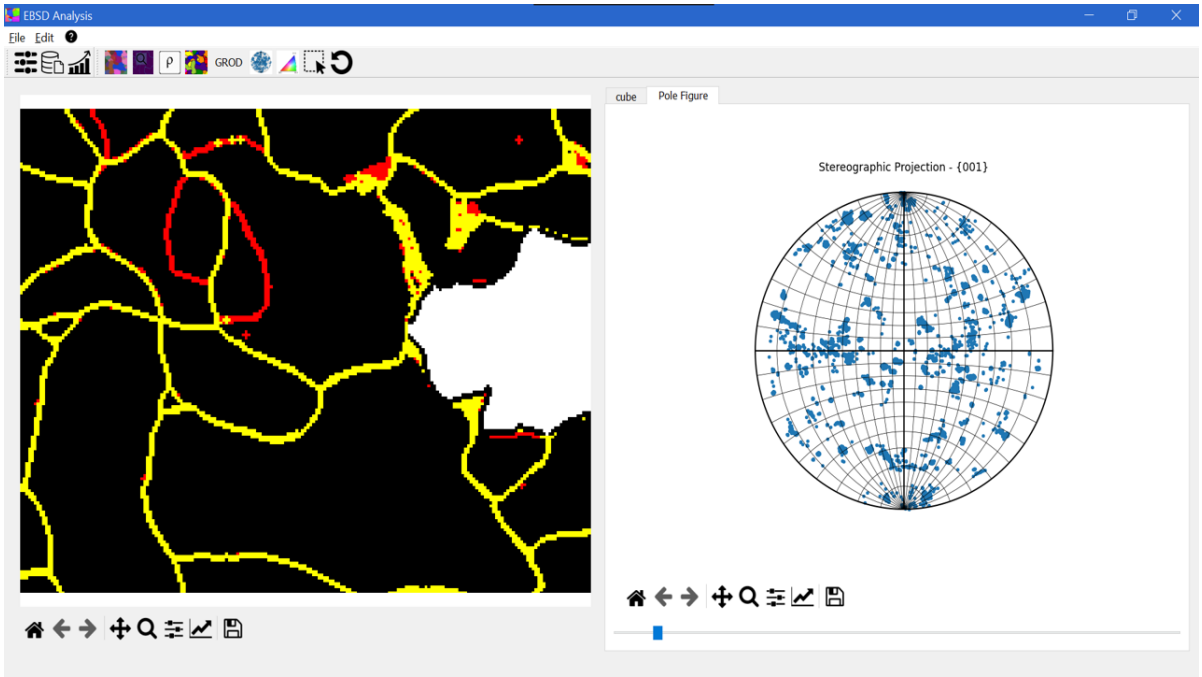


Figure 17 Pole figure for a selected region in the segmented image,

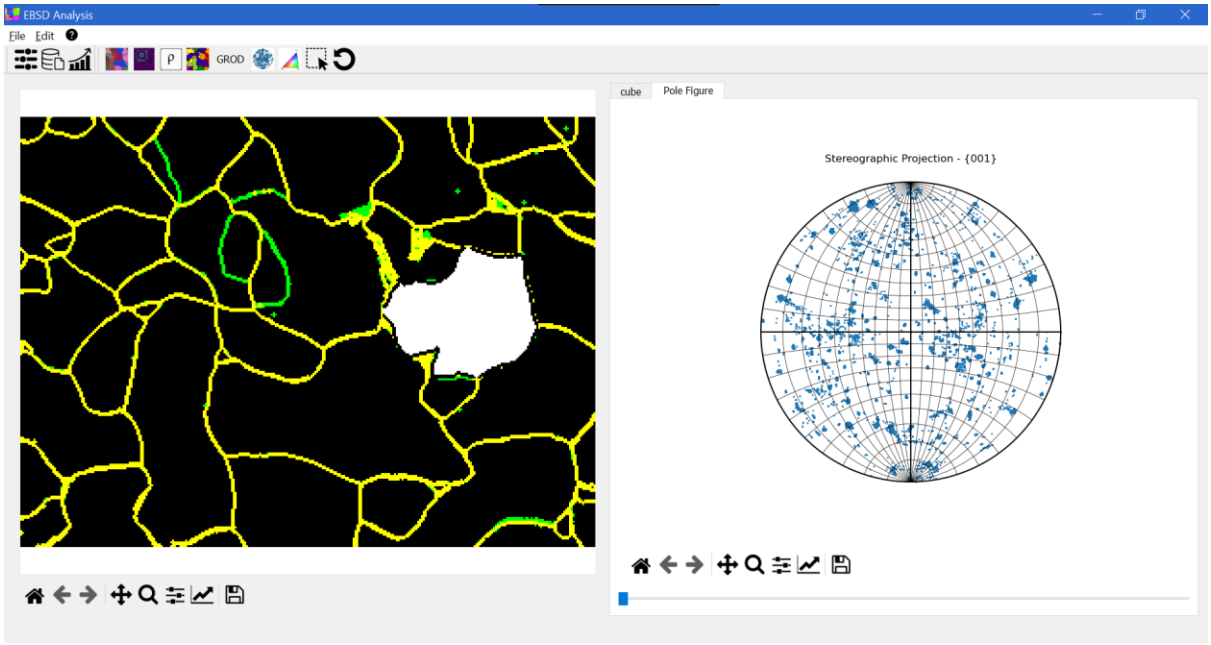


Figure 18 Pole figure for all grain in an EBSD image,

3.3. Inverse Pole Figure

While a pole figure represents a crystal direction or plane normal of a material within the sample reference system, an inverse pole figure displays a specific sample direction within the crystal system. Due to the symmetry of the crystal system in most cases, the inverse pole figure can be reduced to one of the 24 symmetric parts of the pole figure. Thus, the IPF coloring of an EBSD grain image shows which crystal direction is parallel to the sample direction to which the IPF is assigned to [7].

The usual convention is to place red at the <001>, green at the <101> and blue at <111> the directions. Thus, it is simple to visualize for a cubic material that has the face parallel to the set of {001} planes will be in red color, the edge of the crystal face parallel to the set of {101} planes will be in green color and the corner of the crystal face parallel to set of {111} planes will be in blue color. The crystal face intermediate will be given a color which is proportional to these combinations of the three colors, similar to RGB mapping. Typical IPF color map is shown in Figure 19.

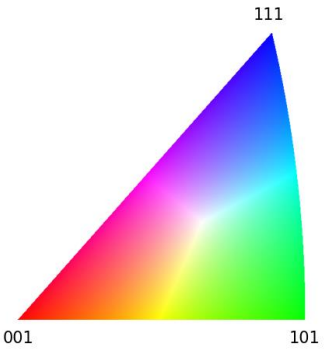


Figure 19 Inverse pole figure color map

3.4. IPF Color Map

The IPF standard colour (Figure 19) obtained for every pixel in the EBSD data image. For any given set of Euler angles $(e_1,\ e_2,\ e_3)$ of a pixel (i, j) obtain the rotation matrix (M). The required axis $[0, 0, 1]$ (in case of IPF-Z) is transformed on multiplying M with the axis which results in a vector say uvw, which is a 3x1 $vector$ and the direction of the point (i, j).

$$M = rotateZXZ(np.identity(3), e_1, e_2, e_3)$$

$$d = [0,0,1]\ for\ IPF - Z$$

$$uvw = M.d$$

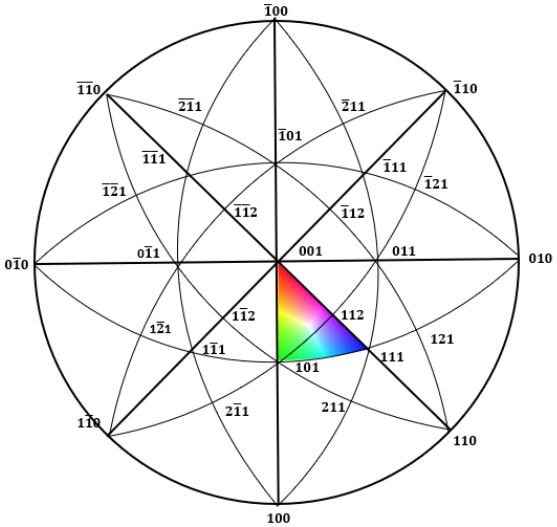


Figure 20 IPF color mapping for the chosen family of directions among the 24 possibilities.

uvw could be any one of the 24 directions in the corresponding family of directions. So, to ensure the direction falls inside the unit triangle region, instead of calculating all the directions for 24 symmetry regions, we can use a simple property of cubic system that is:

The values of *u*, *v*, *w* (of uvw vector) is all positive and

$$w \geq u \geq v$$

Now that uvw falls inside this region, to calculate the corresponding RGB values:

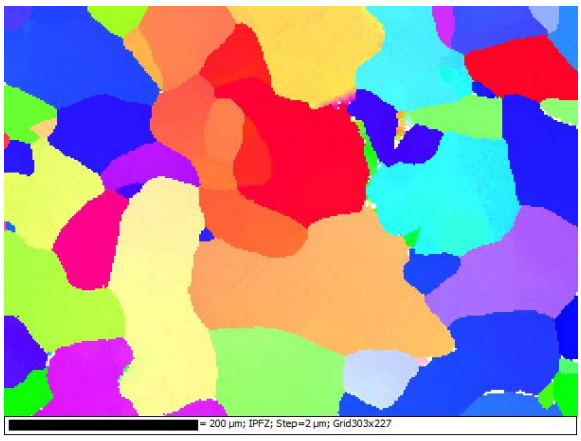
uvw must be the linear combination of the three independent vectors (poles) [001], [101], [111]. On solving the equation, we obtain the values of r, g, b which together gives the color to the pixel point.

$$uvw = r[0,0,1] + g[1,0,1] + b[1,1,1]$$

$$uvw = (g + b,\ b,\ r + g + b)$$

$$r = uvw[2] - uvw[0],\ \ g = uvw[0] - uvw[1] and\ \ b = uvw[1]$$

The 3 inverse pole figures (IPF-X, IPF-Y, IPF-Z) describes the complete orientation for each sampled point in the sample for an easier understanding of crystal orientation. The three different IPF maps have been generated by a Python program and have been compared with the ones generated by a commercial software.



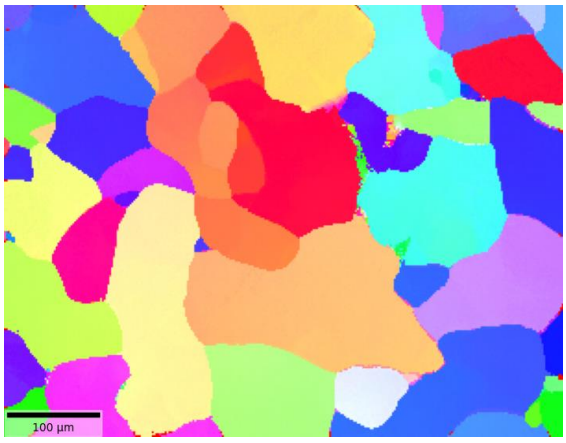


Figure 21 Inverse pole figure generated (IPF-X) generated by a) a commercial software and b) developed in Python.

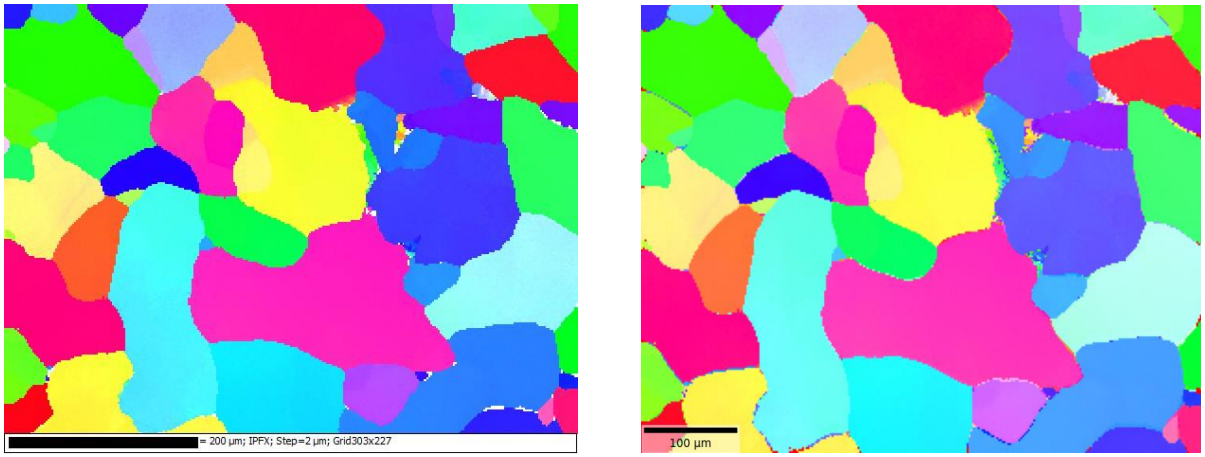


Figure 22 Inverse pole figure generated (IPF-Y) generated by a) a commercial software and b) developed in Python.

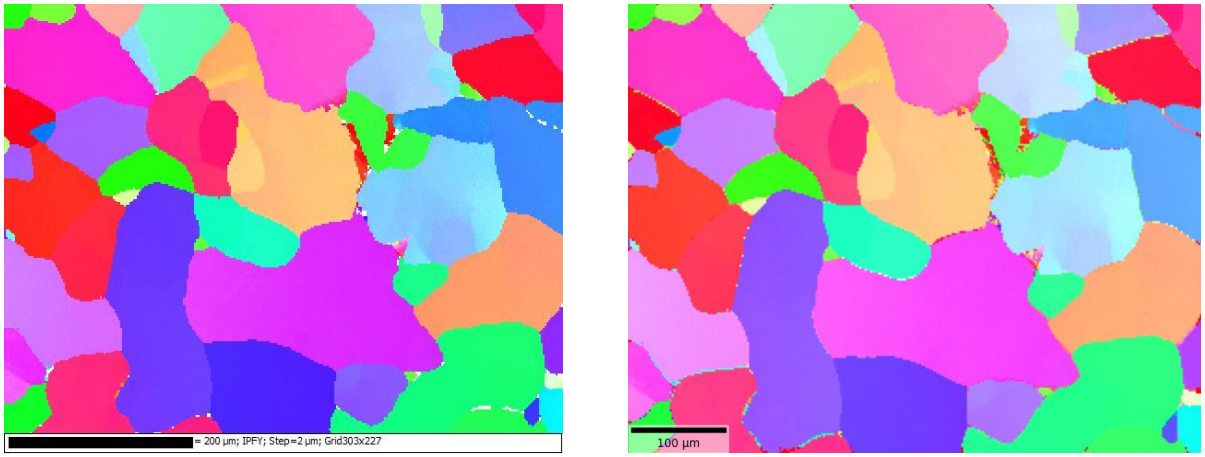


Figure 23 Inverse pole figure generated (IPF-Z) generated by a) a commercial software and b) developed in Python.

Figure 21, Figure 22 and Figure 23 show the IPF-X, IPF-Y, and IPF-Z Maps along with the results of the commercial software. It can be observed that the color maps are exactly matching with one another validating our implementation.

3.5. Density Dislocation

Dislocation in materials, is a crystallographic defect or irregularity with the crystal structure that contains an abrupt change in the arrangement of atoms. The dislocation density, ρ, a type of concentration, is measured by counting the number of dislocation lines that thread a unit area of surface. Dislocation density can be estimated from the computed KAM in EBSD measurement. Figure 24 schematically shows a simple relationship between the misorientation angle (θ), the dislocation spacing, d, and scan step size, L [8].

$$\rho = \frac{\theta}{bL}$$

where, b is the Burgers vector. Considering that $\theta = b/d$ the dislocation density can be estimated

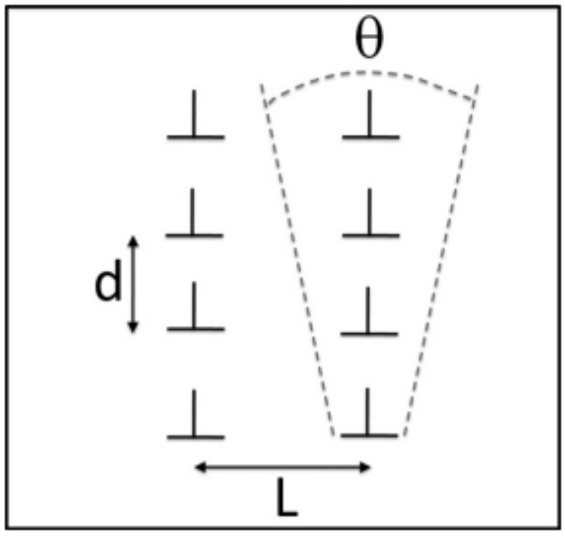


Figure 24 Relationship between the misorientation, h, dislocation spacing, d, and scan step size, L

Figure 25 shows the dislocation density map generated for the ferrite microstructure with the colour bar. The dislocation density as can be seen vary from zero to $2x10^9$.

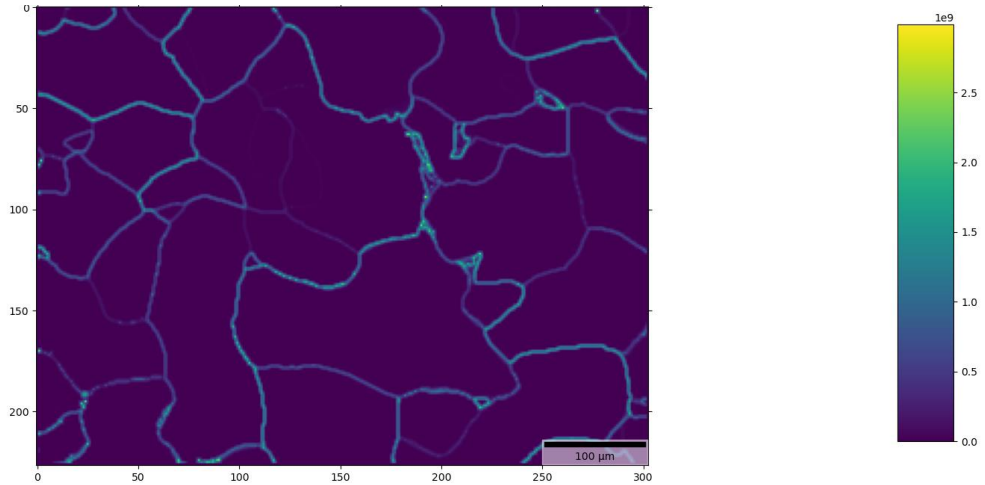


Figure 25 Dislocation density map of the ferrite microstructure based on the computed KAM

3.6. Grain Reference Orientation Deviation

The grain reference orientation deviation (GROD) map provides a measure of misorientation between the mean Euler angles of the grain and the pixels of the individual grains. The

misorientation $GROD_{i,j}$ between the orientation $o_{i,j}$ at position (i,j) and the reference or mean orientation o_g of the grain the position (i,j) belongs to, i.e.,

$$GROD_{i,\ j} = inv(o_g).o_{i,\ j}$$

where, o_g - mean orientation of the grain and $o_{i,\ j}$ - orientation of every pixel in that grain

The generated GROD map for the ferrite microstructure is shown in Figure 26. The computed dislocation density has been compared with the one predicted by the commercial software and the was found to be matching.

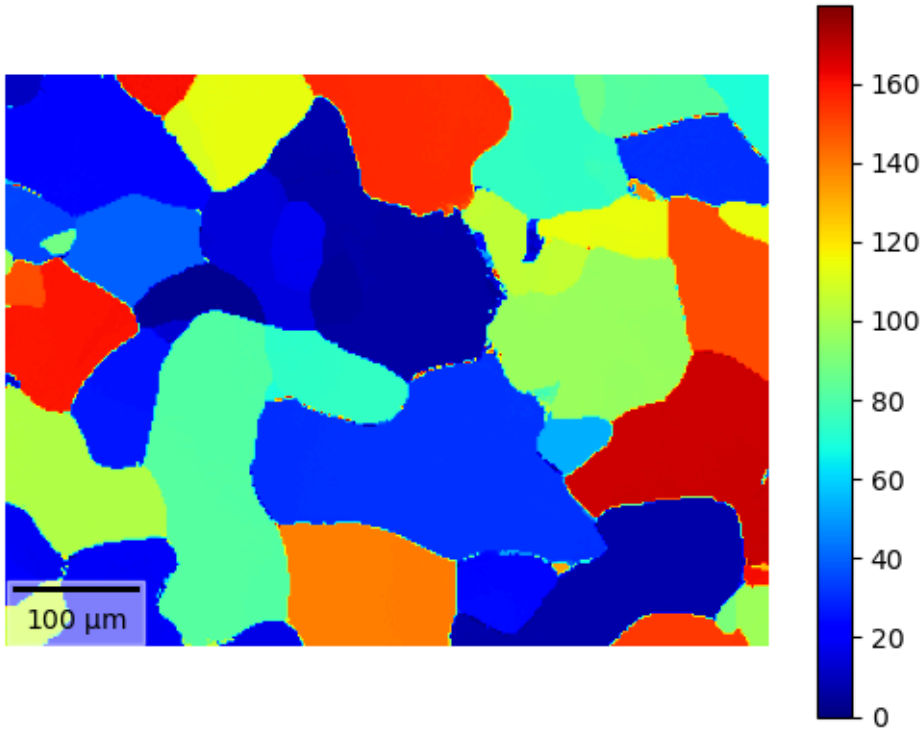


Figure 26 Grain Reference Orientation Deviation (GROD) Map generated using Python program for the ferrite microstructure.

Chapter 4. Features of EBSD Analysis Software

The software essentially consists of Python codes in the form of an import library and PyQt based graphical user interface. The import library contains codes for:

- Reading binary EBSD data to construct RGB images
- Standard region-based segmentation
- Computation of KAM
- KAM based segmentation of images
- Extracting grain properties and
- Generating MAPS such as PF, IPF, dislocation density and GROD

Similarly, the PyQt based GUI contains menu driven interface for the user to interact and execute the python codes.

4.1. General features of the Python codes

The EBSD package could be used as python package by importing the utility functions in the package. The prototypes of the major EBSD library functions and their usage are given in Annexure-I. The Usage example shown in Figure 27 below as a screenshot:

```python
from ebsd import *
import matplotlib.pyplot as plt
print(ebsd.__version__)
print(ebsd.__all__)
fname = 'Data.ctf'
# reading data file
Eulers, data10, cols, rows, xstep, ystep, title = read_data(fname)
# median filtering
Eulers = medianFilter(Eulers, data10, 3)
# rgb image
image = rgb_img(Eulers)
# show image plot
plt.imshow(image)
```

Figure 27 Usage of importing EBSD library in python environment

4.2. General features of GUI

- Cross platform
- Various Segmentation Methods (KAM Based, Watershed)
- Interactive Grain Selection plots
- Inverse Pole Figure and Pole Figure Maps
- Grain Reference Orientation Deviation, Dislocation Density
- Exportable csv file for Grain Details

The main window of the software with available shortcut icons on the front panel are shown and explained in Figure 28.

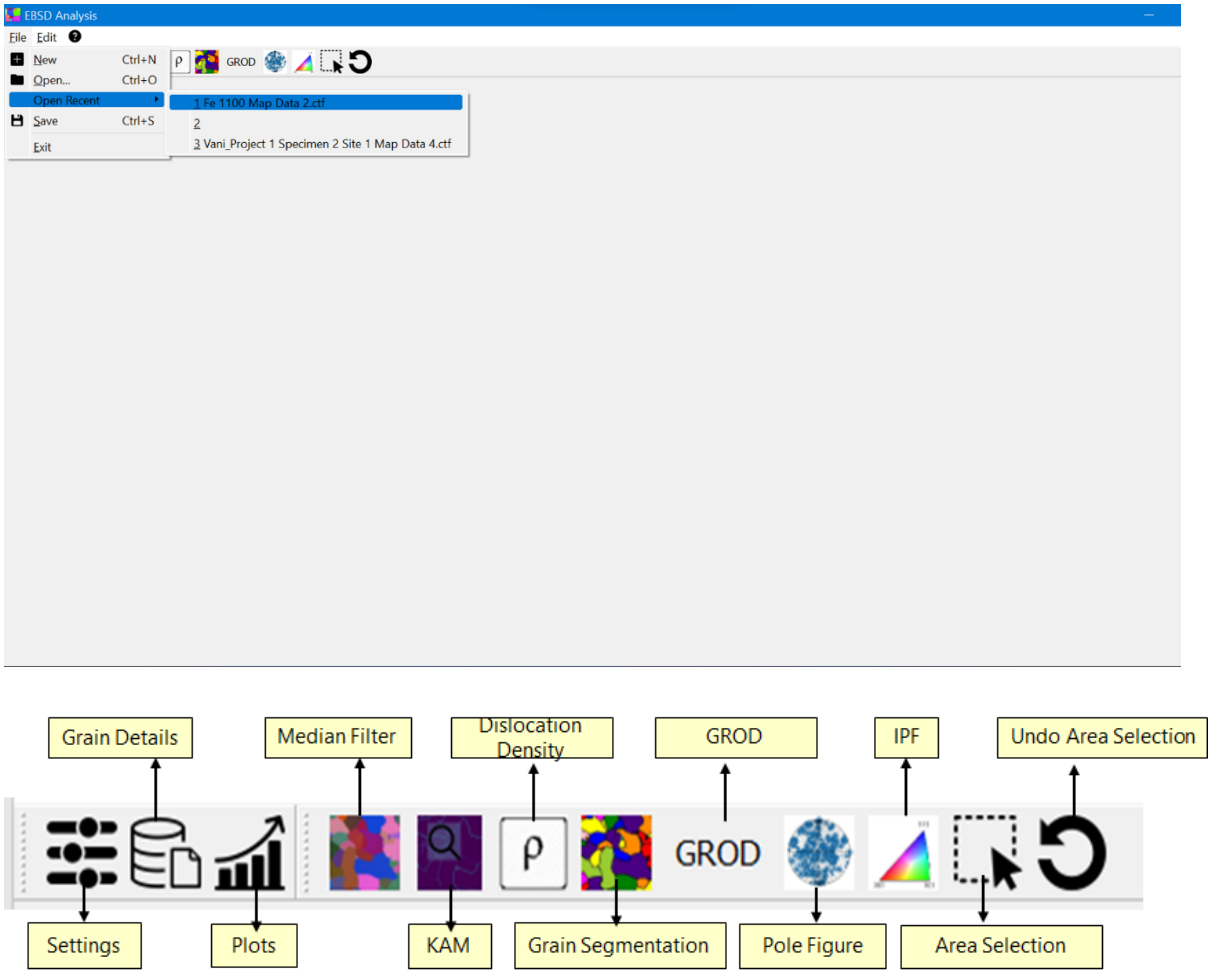


Figure 28 Screenshot of the main window of the software alongwith the explanation of the functionality.

Chapter 5. Conclusion

A dedicated user-friendly software has been developed for segmentation and analysis of EBSD images used for microstructural analysis of materials. The software has been developed using Python language and PyQt GUI development tools. The menu driven GUI enable the user to carry out selection of EBSD data, segmentation of images and generation of various MAPS like the PF and IPF. An EBSD library has been created with functions/methods to carry out various image operations such as KAM, KAM based segmentation, IPF and PF which can be installed and accessed in any Python development environments like the Spyder. The developed software has been used to carry out these operations on different specimens exhibiting different crystal characteristics and has been compared with a commercially available package. The performance has been found to be satisfactory. The scientific community will be richly benefited by this analysis software, which is efficient, benchmarked, and cost-effective.

References

1) B.J. Inkson “Scanning electron microscopy (SEM) and transmission electron microscopy (TEM) for materials characterization”, Scanning electron microscopy (SEM) and transmission electron microscopy (TEM) for materials characterization, Chapter 2.
2) https://en.wikipedia.org/wiki/Otsu's_method
3) Andrew Campbell, Paul Murray, Evgenia Yakushina, Stephen Marshall, William Ion, “New methods for automatic quantification of microstructural features using digital image processing”, Materials and Design, vol. 141, pp. 395–406, 2018.
4) T. Karthikeyan and Saroja Saibaba, “GBgeom: a computer program for visualizing texture parameters and simulating grain boundary structures in cubic crystals”, Journal of Applied Crystallography, vol. 46, pp. 1221–1224, 2013, ISSN 0021-8898.
5) Patrick Camus, “Crystal Orientation Mapping: Orientation Color Scheme”, Technical Note: 51923, Thermo Fisher Scientific, Madison, WI, USA
6) T.B. Britton, J. Jiang, Y. Guo, A. Vilalta-Clemente, D. Wallis, L.N. Hansen, A. Winkelmann, A.J. Wilkinson, “Tutorial: Crystal orientations and EBSD — Or which way is up?”, Materials Characterization, vol. 117, pp. 113–126, 2016
7) T. Karthikeyan, “Area preserving colour coding of inverse pole figure domain”, Journal of Microscopy, vol. 267, no. 1, pp. 107–113, 2017
8) Anish Kumar, and Tresa M. Pollock, “Mapping of femtosecond laser-induced collateral damage by electron backscatter diffraction”, Journal of Applied Physics, vol. 110, no. 083114 pp. 1-5, 2011

Annexure-I- Prototype of major EBSD functions

read_data

---

ebsd.ebsdlib.ebsd.**read_data**(*path_to_file*)

The read_data function takes the file path and returns the Euler angle array for EBSD analysis. It also provides the number of rows, columns and the scan step size of the EBSD data. Currently the package supports reading of any txt file(.ctf, .txt).

Parameters

path_to_file: STR

path of data file in the system.

Returns

Eulers: numpy.ndarray (3,rows,cols)

Euler angle array.

data0: numpy.ndarray

Duplicate of Euler array

cols: int

rows: int

xStep: float

yStep: float

median_filter

---

ebsd.ebsdlib.ebsd.**median_filter**(*Euler_array, data0, w = 5)*

Median Filter function cleans the data values of Euler array with zero values using a kernel of size w and returns the median filtered Euler array.

Parameters

Eulers: numpy.ndarray (3,rows,cols)

data0: numpy.ndarray

w: INT, optional

kernel size. The default is 3.

Returns

Eulers: numpy.ndarray

rgb_img

---

rgb_img function takes the Euler array and returns an array that can represent an image which can be plotted and saved as a .png image.

Returns an Image of all Eulers

Returns

rgb_image: np.uint8 array with rgb values.

find_edges

Given a 2-D image of the grain data find_edges return the edge detected grey image. The sobel operator is used to detect the boundaries.

Parameters

filtered_img: 2D array (0, rows, cols)

Returns

multi: image with edges

binary: binary image with edges.

watersheding

Watershed, an image segmentation technique is used to segment the grains. The function watersheding takes the edge detected image and return a segmented image with labelled regions.

Parameters

multi: image with edges

Returns

wc_img: watersheded image with colored labels.

labels: only labels

n: number of regions identified.

stereographicProjection_001

The stereographic projection gives 3 pole points, for 3 given vectors in 3D space. These pole points are shown in a circular frame. There are 3 different stereographic projections possible {001}, {010} and {100}. Each of them has a different function.

Parameters

qx, qy, qx: tuples

Each tuple represents the normal of any three faces of the cube with orientation same as the grain.

Returns

pf1, pf2, pf3: 3 tuples (x, y) for 3 pole points.

draw_wulff_net

---

ebsd.ebsdlib.ebsd.**draw_wulff_net(ax, step = 9., theta = 0, n = None , **kwargs)**

To draw a wulff net or stereonet for representation of pole figure with default step size of the traces being 9.0

Parameters

ax: matplotlib.pyplost figure axis

step: angle between two adjacent traces

theta: azimuthal angle

n: The number of traces required for covering 180 degrees. Default value is none.

**kwargs: Line properties, optional

color or c: {'k', 'r', 'g', 'b'…}

linewidth or lw: float value

Returns

ax: matplotlib.pyplost figure axis

pf_map

---

ebsd.ebsdlib.ebsd.**pf_map(Eulers, pfch = '{001}')**

Returns a tuple with two list of points pfX and pfY to be plotted in the pole figure map for the given Euler array.

Parameters

Eulers: numpy.ndarray (3,rows,cols)

pfch: string

Option to change the pole axis. Possible values are ‘{100}’, ‘{010}’, ‘{001}’

Returns

pfX: List of x coordinates of the pole points.

pfY: List of y coordinates of the pole points.

kam

---

ebsd.ebsdlib.ebsd.**kam(Eulers)**

Calculates the kernel average misorientation for the given Euler array and returns the KAM image to be displayed.

Parameters

Eulers: numpy.ndarray (3,rows,cols)

Returns

kam: KAM array with values in degrees.

dislocation_density_map

---

ebsd.ebsdlib.ebsd.**dislocation_density(kam, l, b = 235)**

Calculates the Dislocation density from the pre-calculated kernel average misorientation returns an array with dislocation density values.

Parameters

kam: numpy.ndarray(3, rows, cols)

Return value of the kam function.

l: step scan size

xstep or ystep value obtained from read_data function.

b: Burgers vector in Angstrom (10^{-10} m), optional. The default is 235.

Returns

Numpy.ndarray(rows, cols). A 2-D array with dislocation density values in degrees is returned.

ipf

```
ebsd.ebsdlib.ebsd.ipf(Eulers, ipf_ax = 'Z')
```

Calculates the Grain Reference Orientation Deviation using the Euler array and the segmented or labelled grain array.

Parameters

Eulers : numpy.ndarray(3, rows, cols)

ipf_ax : String, optional

The choice for IPF image with respect the axis X, Y, Z. The default is 'Z'.

Returns

ipf_img: numpy.ndarray(rows, cols, 3)

3-D array with rgb color (according to the IPF legend) grain image.

kam_segmentation

```
ebsd.ebsdlib.ebsd.kam_segmentation(Eulers, key = 2.)
```

Grain segmentation based on the KAM values.

Parameters

Eulers : numpy.ndarray(3, rows, cols)

key: float, optional

The minimum angle (in degrees) is considered for segmentation. The default is 2. So any point with misorientation < key will not be considered as separate grain.

Returns

kam_seg: segmented grain image with random colors assigned for labels.

Grains_ids: segmented grain array with integer labels for regions.

ngrains: number of unique regions identified.

grod

---

ebsd.ebsdlib.ebsd.**grod(labels, ngrains, Eulers)**

Calculates the Grain Reference Orientation Deviation using the Euler array and the segmented or labelled grain array.

Parameters

labels: numpy.ndarray(rows, cols)

labels is the grain segmented and labelled array which is return by the segmentation function mentioned above.

ngrains: integer

Number of grains the image is segmented, which is one of the return values of the segmentation functions.

Eulers: numpy.ndarray (3,rows,cols)

Returns

grod_map: numpy.ndarray(rows, cols)

array with GROD values is returned by the function.

ipf_legend

---

ebsd.ebsdlib.ebsd.**ipf_legend(ax = None, n = 512)**

The function draws the IPF color map (or IPF legend) with nxn pixels and a given axis.

ax: matplotlib figure axis, optional.

The IPF legend is plotted in the given axis. If the axis is none, the function plots the IPF legend in a new matplotlib figure.

n: integer, optional

Size of the mesh grid for the IPF legend. The more the value of n is directly proportional to the smoothness of the legend.

Returns

None.