Lars Andreas Hastad Lervik

Orientation and Projection Center Refinement for EBSD Indexing in Python

Master's thesis in Nanotechnology
Supervisor: Jarle Hjelen
Co-supervisor: Håkon Wiik Ånes
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Materials Science and Engineering

NTNU
Norwegian University of
Science and Technology

Lars Andreas Hastad Lervik

Orientation and Projection Center Refinement for EBSD Indexing in Python

Master's thesis in Nanotechnology
Supervisor: Jarle Hjelen
Co-supervisor: Håkon Wiik Ånes
June 2021

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Materials Science and Engineering

Kunnskap for en bedre verden

Preface

This master's thesis is the result of the work done during the spring of 2021 at the Department of Materials Science and Engineering at the Norwegian University of Science and Technology (NTNU). The work is a natural continuation of the project theses: Natlandsmyr [1] and Lervik [2], and is an extension of the open-source Python library *kikuchipy* [3]. The work provides an option to refine the resulting orientations and projection center estimates from electron backscattering diffraction (EBSD) indexing using a wide variety of optimization algorithms in Python. The thesis includes a brief introduction to EBSD theory, dynamical simulations of EBSD patterns, and optimization. I would like to extend my gratitude towards Jarle Hjelen and Håkon Wiik Ånes for their excellent guidance, access to experimental data sets, and contagious curiosity. Would also like to acknowledge the team behind the NTNU IDUN/EPIC computing cluster [4] for providing insight into the scalability of the algorithms presented in this work. I have been standing on the shoulders of giants, and it is my dearest wish that this work can provide meaningful tools to those that follow.

Lars Andreas Hastad Lervik
NTNU, Trondheim
June 11th, 2021

Abstract

In this work a set of algorithms for refining crystal orientations and/or projection center estimates specifically after, but not limited to, dictionary indexing of EBSD patterns are presented. The algorithms are implemented in the Python 3 programming language as an extension of the open-source Python library *kikuchipy*. The algorithms allow for parallel computations and can be run efficiently on a simple laptop, as well as on a computing cluster. Additionally, they have support for handling experimental data sets that can be larger than memory. The refinement can be done with a wide variety of derivative-free optimization methods implemented in the *SciPy* library, both local and global, such as Nelder-Mead and Differential Evolution. The implementation provides reasonable parameters for inexperienced users, while simultaneously providing the ability for customization for more advanced users.

Sammendrag

I dette arbeidet er et sett med algoritmer for forbedring av krystallorientering og projeksjonssenter estimater spesielt etter, men ikke begrenset til, dictionary indexing av EBSD mønster presentert. Algoritmene er utviklet i programmeringsspråket Python 3 som en utvidelse av open-source Python-biblioteket *kikuchipy*. Algoritmene kan gjøre beregninger parallelt, og kan kjøres effektivt på en bærbar datamaskin, samt på dataklynger. I tillegg har algoritmene støtte for å arbeide med eksperimentelle datasett som er større enn tilgjengelig minne. Forbedringen kan gjøres med et vidt utvalgt av derivatfrie optimaliseringsalgoritmer som er implementert i *SciPy* biblioteket, både lokale og globale, som for eksempel Nelder-Mead og Differential Evolution. Implementasjonen tilbyr rimelige parametere for uerfarne brukere, samtidig som den tillater erfarne brukere å finpusse på parametere.

Contents

Preface . . . . **i**
Abstract . . . . **iii**
Sammendrag . . . . **v**
Contents . . . . **vii**
Figures . . . . **ix**
Tables . . . . **xiii**
Code Listings . . . . **xv**
1 Introduction **1**
1.1 Background and Motivation 1
1.2 Aim and Scope of the Work 2
2 Theoretical Background **3**
2.1 Electron Backscatter Diffraction 3
2.2 Indexing of EBSD Patterns 5
2.2.1 Hough Indexing 5
2.2.2 Pattern Matching 5
2.3 Optimization 9
3 Software Tools **11**
3.1 Python Package Dependencies 11
3.1.1 NumPy 11
3.1.2 SciPy 11
3.1.3 Dask 12
3.1.4 Numba 12
3.1.5 orix 12
3.2 kikuchipy 12
3.3 EMsoft 12
4 Algorithms **15**
5 Experimental **17**
5.1 Refinement Performance Metrics 18
5.1.1 Testing of Various Optimization Algorithms 18
5.1.2 Effects of Inaccurate Projection Center Estimates 18
5.1.3 Effect of Improved Resources 19
5.2 Nickel Acquired with Increasing Camera Gain 19
5.3 Simulated Large Scan of Nickel 19
5.4 Single Crystal Silicon Wafer 22

6 Results and Discussion **25**
6.1 Refinement Performance Metrics 25
6.1.1 Testing of Various Optimization Algorithms 25
6.1.2 Effects of Inaccurate Projection Center Estimates 25
6.1.3 Effect of Improved Resources 26
6.2 Nickel Acquired with Increasing Camera Gain 30
6.3 Simulated Large Scan Ni 35
6.4 Single Crystal Silicon Wafer 36
6.4.1 Simulated Large Scan Si 36
6.4.2 Comparison of Orientation Sampling Tools 37
7 Conclusion **41**
8 Further Work **43**
Bibliography **45**
A Refinement Implementation **51**
A.1 Calling Functions 51
A.1.1 Full Refinement 51
A.1.2 Orientation Refinement 54
A.1.3 Projection Center Refinement 57
A.2 Solver Functions 60
A.2.1 Full Refinement 60
A.2.2 Orientation Refinement 61
A.2.3 Projection Center Refinement 62
A.3 Objective Functions 63
A.3.1 Full Refinement 63
A.3.2 Orientation Refinement 64
A.3.3 Projection Center Refinement 65
A.4 Single Pattern Simulation Functions 66
A.4.1 Compute Single Pattern 66
A.4.2 Master Pattern Data Extraction 67
A.4.3 Computation of Direction Cosines with Multiple Projection Centers 68
A.4.4 Computation of Direction Cosines with a single, fixed Projection Center 69
A.4.5 Lambert Projection of Direction Cosines 69
A.4.6 Lambert Projection Interpolation Parameters 70
A.5 Similarity Metrics 71
A.5.1 Normalized Cross Correlation Coefficient 71
A.5.2 Normalized Dot Product 71
B Code Examples **73**
B.1 Scalability Evaluation Script 73
B.2 Effect of Inaccurate Projection Center 74
B.3 Projection Center Correction Step 76
B.4 Orientation Correction Step 76
B.5 scikit-image Mask from Experimental Pattern. 78

Figures

2.1 Schematic of an experimental setup during an EBSD scan with different EBSD patterns for each scan location in the region of interest. Adapted from [14]. 4
2.2 EBSD pattern from a single crystal silicon scan at 20 keV. 4
2.3 Hough transform of three points on a straight line. Points on the same straight line, intersect in Hough space and create peaks of high intensity. 5
2.4 Roşca-Lambert and stereographic projection of the silicon master pattern simulated in EMsoft 5.0. 6
2.5 Schematic of the experimental geometry used in the simulation of EBSD patterns. From [6]. 7
2.6 Comparison of Hough Indexing and Dictionary Indexing with increasing levels of camera gain, ranging from 0 dB to 22 dB, on recrystallized nickel. Adapted from [10]. 9

4.1 Example task graph of the refinement process of three experimental patterns. The refinement of each pattern does not need to communicate with each other, highlighting the embarrassingly parallel nature of the optimization problem. 15
4.2 Description of single pattern refinement, where the parameters that are optimized could be the Bunge-Euler angles, projection center coordinates, or both. 16

5.1 m-3m IPF color key. 17
5.2 Nickel master pattern at 20 keV, in the Roşca-Lambert projection simulated in EMsoft 5.0 20
5.3 Simulated nickel pattern at the center of the simulated large scan, with the projection center marked with a yellow marker. 22
5.4 Simulated nickel patterns at the corners of the simulated large scan, with the original projection center marked with a yellow marker. . . 23
5.5 EBSD pattern at location 23, 24 in the Si scan before and after pattern processing in kikuchipy. 24
5.6 Experimental and simulated EBSD patterns from the Si scan after they have had a mask applied to them. 24

6.1 Mean NCC-score after refinement of DI results with varying percent error in the projection center parameters used during indexing. . . 27
6.2 Plot of the change in runtime from running the orientation refinement on the NTNU IDUN [4] cluster with increasing number of CPU cores. 28
6.3 Plot of Amdahl's law for different p-values, together with the speedup achieved from running the orientation refinement on the NTNU IDUN [4] cluster with increasing number of CPU cores. 29
6.4 Effects of removing the static and dynamic background, and averaging each pattern with its nearest neighbors using a (3×3) Gaussian window in kikuchipy for different scans with increasing gain, at the same location. 30
6.5 IPFs of kikuchipy DI results using orix sampling. 30
6.6 IPFs of refined kikuchipy DI results using orix sampling. 31
6.7 Gaussian kernel estimation, with a smoothing factor of 0.2, of the PDF to the NCC-scores in the different scans with orix sampling. . . 31
6.8 Scan 1 after DI in kikuchipy with orix sampling, where the NCC scores are added on top of the image. The red circles highlight areas with very low NCC scores, indicating poor matches. 32
6.9 IPFs of kikuchipy DI results using EMsoft sampling. 32
6.10 NCC-score difference for DI results for scan 1, using EMsoft sampling and orix sampling. 33
6.11 IPFs of refined kikuchipy DI results using EMsoft sampling. 33
6.12 Gaussian kernel estimation, with a smoothing factor of 0.2, of the probability density function to the NCC-scores in the different scans with EMsoft sampling 34
6.13 Misorientation angle in degrees, to the central pixel in the simulated large Ni scan. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead. 35
6.14 Misorientation angle in degrees, capped at 10°, to the central pixel in the Si wafer scan with orix sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead. 36
6.15 Misorientation angle in degrees, capped at 10°, to the central pixel in the simulated large Si scan with orix sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead. 37

6.16 Misorientation angle in degrees, capped at 10°, to the central pixel in the simulated large Si scan with EMsoft sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead. 38
6.17 Degree of misorientation, capped at 10°, to the central pixel in the Si wafer scan with EMsoft sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead. 39
6.18 The EBSD patterns being used in the DI routine at the top left and center of the scan, compared to the orientation that matched using DI with EMsoft sampling. 40

Tables

2.1 Four-point interpolation weight assignment. 8

5.1 Acquisition parameters for the nickel data set, with the projection center, (x^*, y^*, z^*), in TSL convention [10]. 19
5.2 Camera gain and exposure time, t, for the different nickel scans. Adapted from [10]. 19
5.3 Crystal data parameters used in the DI routine. 20
5.4 Monte Carlo and master pattern simulation parameters. 21
5.5 Parameters for simulated, single crystal, large scan of nickel. Projection center, (x^*, y^*, z^*), is given in TSL convention. 22
5.6 Acquisition parameters for the single crystal silicon data set, with the projection center, (x^*, y^*, z^*), in TSL convention. 23

6.1 Performance of various optimization algorithms, on an Intel i7-4790 CPU @ 3.60 GHz, attempting to refine the built in nickel data set in kikuchipy. 26

Code Listings

A.1 Required imports for the code listings presented in this chapter. . . 51
A.2 Full refinement method. 51
A.3 Orientation refinement method. 54
A.4 Projection center refinement method. 57
A.5 Full refinement solver method. 60
A.6 Orientation refinement solver function. 61
A.7 Solver function for projection center refinement. 62
A.8 Objective function for the full refinement. 63
A.9 Objective function for the orientation refinement. 64
A.10 Projection center refinement objective function. 65
A.11 Method to simulate a single EBSD pattern. 66
A.12 Method to extract data from master patterns. 67
A.13 Method to calculate direction cosines when the experimental geometry has multiple projection centers. 68
A.14 Method to calculate direction cosines when the experimental geometry has a single, fixed projection center. 69
A.15 Project a vector to the Lambert projection. 69
A.16 Fast method to get the Lambert interpolation parameters. 70
A.17 Quick NCC calculation. 71
A.18 Quick NDP calculation. 71
B.1 Script to time the performance of the orientation refinement. 73
B.2 Simulation of ground truth. 74
B.3 DI with erorr in projection center parameters, followed by refinement. 75
B.4 Projection Center correction factor method. 76
B.5 Orientation correction factor method. 76
B.6 Creation of threshold mask using scikit-image. 78

Chapter 1

Introduction

1.1 Background and Motivation

Electron backscatter diffraction (EBSD) is a characterization technique for polycrystalline materials utilizing a scanning electron microscope (SEM). Its ability to routinely and reliably determine individual grain properties, such as grain orientation and phase, has lead to its popularity in materials science and geology [5]. A typical experimental setup consists of a stationary beam of high energy electrons, around 20 keV, which scans a sample that is tilted 70° with respect to the horizontal. The sample tilt is in place to ensure that the EBSD patterns on the detector are of sufficient intensity. The EBSD patterns are a result of the high energy electrons in the SEM chamber entering the sample, with an interaction volume around 20 nm beneath the target point, which backscatter and diffract [5]. The electrons' path will depend on the crystal lattice and its orientation, thus the EBSD patterns represent a gnomonic projection of the crystal lattice [5, 6]. This property of EBSD patterns allows for indexing, where one can label the experimental EBSD patterns with unit cell orientations and phase identity.

There are two general strategies for EBSD pattern indexing, Hough indexing (HI), and dynamical simulation based pattern matching methods, first described by Winkelmann *et al.* [7]. HI attempts to localize bands in the EBSD patterns by transforming these to peaks in Hough space, which are easy for a computer to localize. By calculating the angles between the bands and comparing these to a lookup table based on the crystal structure, one can index the EBSD patterns [5, 8]. The effectiveness of HI is entirely based around its ability to detect bands. Thus, if the EBSD patterns contain a lot of noise, or there are bands from a neighboring grain present, the indexing success rate for HI will suffer [9, 10]. Pattern matching methods attempt to match a dynamical simulated pattern with the experimental EBSD pattern. In this work the dictionary indexing (DI) approach [11] has been the pattern matching method of choice. Image processing techniques which compare all the pixels in the pattern are used to determine the similarity between the simulated patterns in the dictionary, consisting of patterns with uniformly sampled orientations from the Rodrigues fundamental zone, and the experimental EBSD

pattern. A high similarity is a good indication that there is a match. Naturally, in order for DI to provide good indexing results, a sufficiently large dictionary of EBSD patterns needs to be simulated. However, due to the discrete nature of the dictionary, DI only provides reasonable initial guesses for the orientations and an orientation refinement is required [9].

1.2 Aim and Scope of the Work

The aim of this work has been to develop an open-source refinement module, with scalable parallelism, for kikuchipy [3], an open-source Python library for processing and analysis of EBSD patterns, that can improve the crystal orientations and the projection center estimates of EBSD indexing results. Accessibility has been a core value during this work and the refinement process should run efficiently on a simple laptop and large clusters while handling experimental data that can be larger than memory. Furthermore, the barrier to entry should be as low as possible, such that an inexperienced user can get very reasonable refinement results with no knowledge about the process. This is achieved by providing reasonable default parameters. However, the implementation should also be customizable in order for the advanced user to fine-tune the refinement parameters for optimal results. Refinement is the final step of the kikuchipy DI routine implemented in [1] and [2].

Chapter 2

Theoretical Background

The theoretical background in this work builds on the theoretical background in the author's project thesis[2].

2.1 Electron Backscatter Diffraction

EBSD is a SEM based characterization technique for polycrystalline materials, suited to determine grain orientations and phase. An experimental schematic is shown in Figure 2.1. A stationary beam of high energy electrons, around 20 keV, scans an area of the sample with a step size of around 10-1000 nm with one EBSD pattern per scan point. The working distance between the objective lens and the sample is usually around 10-30 mm. The sample is tilted approximately 70° with respect to the horizontal in order to increase the EBSD pattern intensity on the detector, which usually is orthogonal to the incident beam direction [5]. The detector traditionally consisted of a phosphor screen connected to a charge-coupled device (CCD) camera, however, there are newer detectors that can directly detect backscattered electrons leading to sharper EBSD patterns [12, 13]. The EBSD patterns typically range from 50-500 pixels in both the x- and y-direction. The EBSD patterns represent the gnomonic projection of the Kikuchi sphere, which can be thought of as a unique map of the backscattered electron yield for a crystal lattice [5, 6]. The theory around the creation of EBSD patterns is based on the Bloch wave theory of electron diffraction, described in [7]. Simplified, the crystal lattice and the incidence angle of the incoming electrons, relative to the crystal planes, channel electrons further in certain directions. Together with the variable probability of inelastic scattering due to diffraction, this makes up the general theory of EBSD patterns consisting of inelastic scattering and both incident and outgoing diffraction [6]. Figure 2.2 shows an EBSD pattern of silicon. The pattern is constructed of Kikuchi bands, which are contained within Kikuchi lines representing Kossel cones. The angular width of the Kikuchi bands is twice the Bragg angle, θ_{hkl}. From Bragg's law

$$2 \cdot d_{hkl} \cdot \sin\theta_{hkl} = n \cdot \lambda \tag{2.1}$$

where d_{hkl} is the interplanar spacing, n is the order of reflection, and λ is the wavelength of the incident electron beam, one can relate the width of the bands to the interplanar spacing. The geometric projection is represented by the center of the bands, and the interplanar angles can be found by comparing the angle between geometric projections [5].

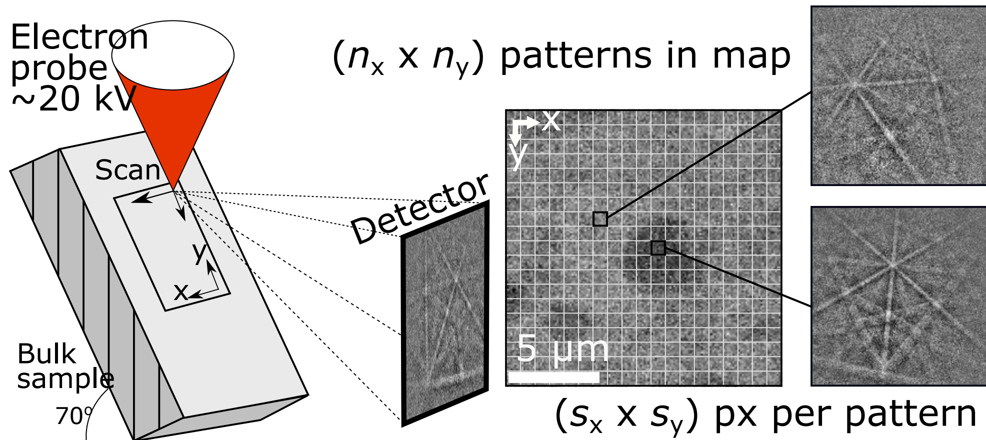


Figure 2.1: Schematic of an experimental setup during an EBSD scan with different EBSD patterns for each scan location in the region of interest. Adapted from [14].

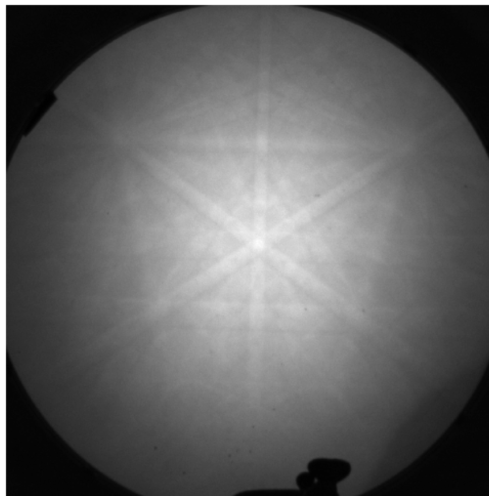


Figure 2.2: EBSD pattern from a single crystal silicon scan at 20 keV.

2.2 Indexing of EBSD Patterns

2.2.1 Hough Indexing

HI is a quick automated indexing routine, which gets its name due to the Hough transformation of the EBSD pattern. The algorithm is described in detail in [8]. The transformation makes it easier for a computer to localize straight lines, as these become peaks in Hough space. The polar equation of a straight line is given by:

$$\rho = x \cos \theta + y \sin \theta \tag{2.2}$$

where ρ is the distance of the line from the origin, and θ is the angle between the x-axis and the normal from the origin to the line. [5]

Figure 2.3 shows the transformation of three points of a straight line. The overlap in parameter space is stored as a peak representing a Kikuchi band. After the transformation, interplanar angles and spacings are calculated and compared against lookup-tables for each of the candidate phases. The best fitting values are then assigned to each of the experimental EBSD patterns in the scan.

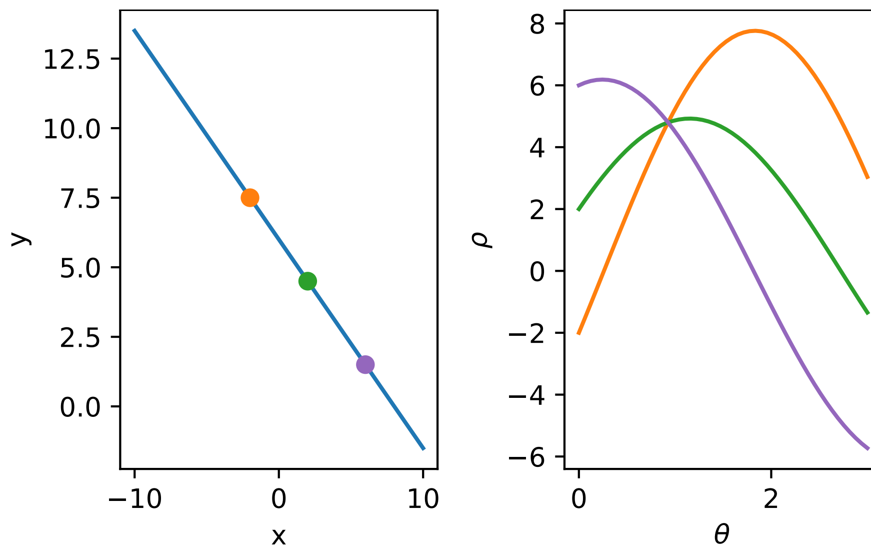


Figure 2.3: Hough transform of three points on a straight line. Points on the same straight line, intersect in Hough space and create peaks of high intensity.

2.2.2 Pattern Matching

The newer approach to EBSD indexing is based around pattern matching utilizing dynamical simulations, first described by Winkelmann *et al.* [7]. Several pattern matching implementations exist, such as EMsoft [15], AstroEBSD [16], and kikuchipy [3]. In this work when dealing with pattern matching, a method called dictionary indexing (DI) is used. This work is an extension to the kikuchipy implementation, which is based on the EMsoft implementation described by Callahan

and De Graef [6]. The central component of DI is the dynamical simulation of master patterns prior to indexing. The master patterns are the 2D representation of the Kikuchi sphere, which can be thought of as enclosing a unit cell within a unit sphere and storing the backscattered electron yield for any given exit direction. A method for mapping uniform grids on a sphere, and the inverse, was presented by Roşca [17]. The inverse mapping, described by Equation (2.3) and Equation (2.4), can transform a Cartesian coordinate, (x, y, z), and map it to a point (X, Y) on the 2D, square, equal-area Roşca-Lambert projection.

$$(X,Y) = \mathrm{sign}(x)\sqrt{2(1-z)}\left(\frac{\sqrt{\pi}}{2}, \frac{2}{\sqrt{\pi}}\arctan\frac{y}{x}\right), |y| \leq |x| \tag{2.3}$$

$$(X,Y) = \mathrm{sign}(y)\sqrt{2(1-z)}\left(\frac{2}{\sqrt{\pi}}\arctan\frac{x}{y}, \frac{\sqrt{\pi}}{2}\right), |x| \leq |y| \tag{2.4}$$

Thus, it is possible to map each of the Kikuchi hemispheres down to the 2D-plane in the form of the northern and southern master patterns. If the candidate phase is centrosymmetric, the hemispheres will be identical and it is possible to represent the full Kikuchi sphere with only one of the hemispheres. The master pattern for silicon, simulated in EMsoft 5.0, is shown in Figure 2.4.

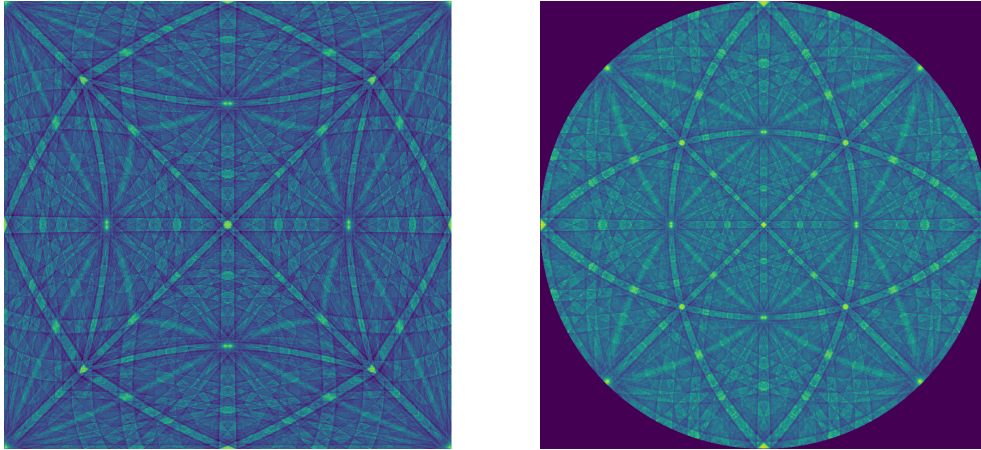


Figure 2.4: Roşca-Lambert and stereographic projection of the silicon master pattern simulated in EMsoft 5.0.

To simulate EBSD patterns, a model of the experimental geometry is required. A schematic of the experimental model is shown in Figure 2.5. In the model the sample is located in the EDAX TSL (RD, TD, ND) sample reference frame, with the origin located at the top left corner of the sample. The detector reference frame follows the right hand rule and is defined as x_d having direction opposite to TD, y_d directed upwards towards the pole piece, and z_d normal to the detector screen. θ_c represents the detector tilt, and σ represents the sample tilt with respect to the horizontal plane. L is the distance between the detector screen and the interaction point P on the sample. λ is the distance between the detector screen and the point

of intersection between the detector axis and incident beam direction, Q, which is located a distance ζ_c below the pole piece. W describes the working distance between the pole piece and the interaction point. The projection/pattern center is defined as the location on the detector screen with the shortest distance to the interaction point. Changing the length of L by ΔL does not alter the projection center. The change results in a zoom around the projection center coordinates, (x_{pc}, y_{pc}), that is inverse to ΔL.

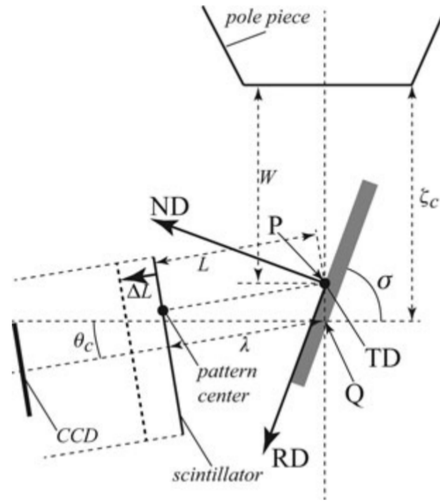


Figure 2.5: Schematic of the experimental geometry used in the simulation of EBSD patterns. From [6].

By creating vectors between an arbitrary point on the detector screen, $(x_d, y_d, 0)$ in the detector reference frame, to the interaction point P, and modifying the vector to be in the sample reference frame, one has obtained inverse electron channeling directions between the detector and sample. It has been shown by Callahan and De Graef [6] that the detector screen coordinates can be represented in the sample reference frame by the following equation:

$$\mathbf{r}_g = [(y_{pc} - y_d) \cdot \cos\alpha + L \cdot \sin\alpha, x_{pc} - x_d, -(y_{pc} - y_d) \cdot \sin\alpha + L \cdot \cos\alpha] \quad (2.5)$$

where the interaction point, P, is represented by the coordinates (x_{pc}, y_{pc}, L) in detector reference frame. α is defined as the angle between the ND-axis and the detector normal, given by:

$$\alpha = \frac{\pi}{2} - \sigma + \theta_c \quad (2.6)$$

with σ and θ_c defined as above. The length of $\mathbf{r}_g$ is given by:

$$|\mathbf{r}_g| = \sqrt{L^2 + (y_{pc} - y_d)^2 + (x_{pc} + x_d)^2} \quad (2.7)$$

The direction cosines from an arbitrary point on the detector screen, (x_d, y_d), to the interaction point, P, can thus be described in the sample reference frame by:

$$\hat{\mathbf{r}}_g(x_d, y_d) = \frac{\mathbf{r}_g}{|\mathbf{r}_g|} \tag{2.8}$$

The direction cosines for every point on the detector screen are then mapped to points on the master pattern using Equation (2.3) and Equation (2.4). The EBSD patterns are then created using a four-point interpolation of the mapping. For every point on the EBSD pattern, the direction cosines to the corresponding point (x_d, y_d) on the detector screen and its neighbors, (x_d+1, y_d), (x_d, y_d+1), (x_d+1, y_d+1), are given weights depending on how close they are to integer values. As an example the mapping $(X, Y) = (10.3, 7.5)$ would create the weights $a = 0.3$, $b = 0.5$ and $c = 0.7$, $d = 0.5$. The weight assignment is shown in Table 2.1.

Table 2.1: Four-point interpolation weight assignment.

Point	Weights
(x_d, y_d)	$c \cdot d$
$(x_d + 1, y_d)$	$a \cdot d$
$(x_d, y_d + 1)$	$c \cdot b$
$(x_d + 1, y_d + 1)$	$a \cdot b$

To simulate an EBSD pattern with for a rotated crystal, the initial direction cosines for the experimental geometry needs to be rotated, for example by a unit quaternion, and the the mapping and interpolation can be done as above. However, the steps detailed above is just to simulate a single EBSD pattern, and as the name implies, DI requires a dictionary or a collection of simulated EBSD patterns to match against the experimental EBSD patterns. The Rodrigues fundamental zone (RFZ) is a polyhedron in Rodrigues space, whose shape is governed by the crystal symmetry of the material, which contains all the unique orientations of the unit cell [18]. Thus, in order to create a sufficient dictionary for indexing, a uniform sampling of the RFZ is required. Naturally, the sampling will be discrete with a specific angular step size. In reality, the orientations are somewhere in-between the discrete values and DI without orientation refinement only provides reasonable initial guesses, and there is a need for a refinement step [9].

After the dictionaries have been created for each of the candidate phases, every experimental EBSD pattern is compared to every simulated EBSD pattern using image processing techniques, such as the normalized dot product (NDP) [19] or the normalized cross correlation coefficient (NCC) [20], to determine the similarity between the two patterns. The orientations for the simulated patterns with the highest similarity metric are stored and used to index the experimental pattern. The main difference between HI and DI comes from the fact that DI utilizes every pixel in the pattern to do the indexing, while HI relies on band localization. As a consequence, with increasing levels of noise in the experimental data, the

process of localizing bands becomes harder. Figure 2.6 shows the indexing results of both HI and DI with increasing levels of camera gain on recrystallized nickel. As the noise in the experimental pattern increases, so does the difficulty in localizing bands and HI simply collapses, while DI will still assign the best matching orientation in the dictionary, regardless how low the similarity metric is.

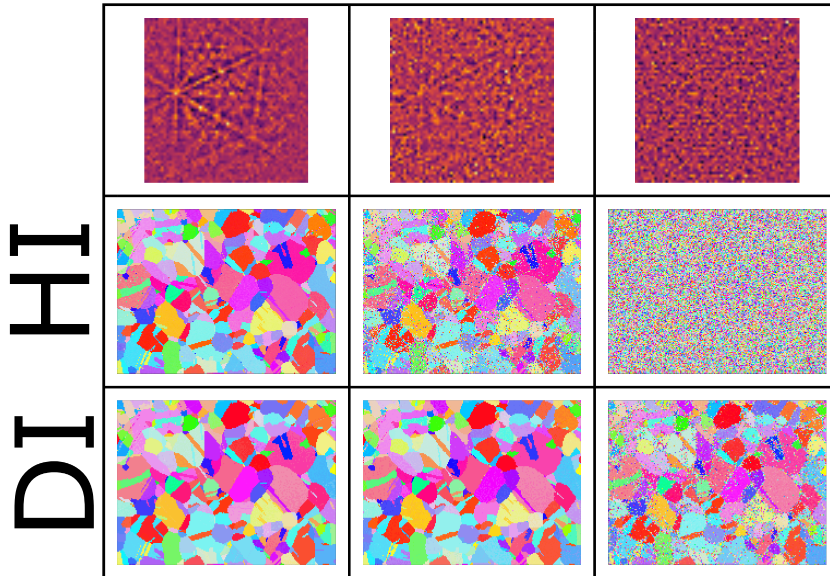


Figure 2.6: Comparison of Hough Indexing and Dictionary Indexing with increasing levels of camera gain, ranging from 0 dB to 22 dB, on recrystallized nickel. Adapted from [10].

2.3 Optimization

Optimization is the process of maximizing or minimizing an objective function value $f(\mathbf{x})$ in D dimensions, where $\mathbf{x} \in \mathbb{R}^D$ [21]. The properties of the objective function and the sufficiency of local optimums determine the suitable optimization methods for the problem. The forward model used to simulate EBSD patterns in this work does not have an analytical expression as a function of projection center and crystal orientation, thus, derivative free optimizers (DFOs) are required to solve the optimization problem which wants to maximize the similarity between simulated and experimental patterns [9]. The implementation in this work supports the following local DFOs: Nelder-Mead [22, 23], modified Powell's algorithm [24], and the stochastic, global DFOs: Basin-hopping [25], Dual Annealing [26], and Differential Evolution [27].

In this work there are three different optimization problems for the refinement process. Two are in three dimensions, where either the projection center coordinates (x^*, y^*, z^*) or the orientation represented by the Bunge-Euler angles (ϕ_1, Φ, ϕ_2) are to be optimized. The third combines the parameter spaces and ends up having a six dimensional space where $(\phi_1, \Phi, \phi_2, x^*, y^*, z^*)$ are to be optimized. Pang *et al.* [28] has described the optimization landscape in detail, and highlighted the fact that for $\mathbf{x} \in \mathbb{R}^6$ the parameters interfere with each other, creating a *sloppy* optimization landscape where a solution to the optimization problem, can in reality be incorrect.

Chapter 3

Software Tools

The majority of this work was done in Python 3, which has become an increasingly popular programming language, partly due to the enormous Python Package Index (PyPI) with over 309 000 projects available for download [29]. This chapter attempts to shine a light on the packages used in this work which have been crucial for the implementation and performance of the algorithms.

3.1 Python Package Dependencies

The following Python packages are used inside of the refinement algorithms presented in this work.

3.1.1 NumPy

NumPy is an open-source Python library that provides homogeneous, in-memory, multidimensional Python array objects operating on the CPU [30]. It is at the core of numerical computations in Python, and all of the following Python packages use NumPy under the hood, or in the case of another Python package, Numba, is able to translate it to efficient machine code. In this work NumPy 1.19.2 was used.

3.1.2 SciPy

SciPy is an open-source Python library with a collection of user-friendly and efficient numerical algorithms for numerical integration, interpolation, linear algebra, statistics, and most importantly for this work, optimization [31]. The optimization module provides the refinement process with all minimization methods used in the implementation. In this work SciPy 1.5.2 was used.

3.1.3 Dask

Dask is an open-source Python library for parallel computing [32]. The task of refining dictionary indexed EBSD data falls under a category known as *embarrassingly parallel* workloads, Furthermore, Dask allows the code to scale up to large clusters, while still running efficiently on a simple laptop. The library also facilitates working on larger-than-memory data sets. In this work Dask Delayed was used to make the algorithms, which otherwise would run sequentially, run in parallel. In this work Dask 2.30.0 was used.

3.1.4 Numba

Numba is a an open-source just-in-time compiler, which compiles code at run time, and translates a subset of Python and NumPy code into efficient machine code with comparable performance to compiled languages such as C and FORTRAN [33]. As the refinement is an iterative process, Numba has been an important part in getting the runtime of the process down to an acceptable state. In this work Numba 0.51.2 was used.

3.1.5 orix

orix is an open-source Python library for analysing orientations and crystal symmetry [34, 35]. In the refinement algorithms orix' CrystalMap class is used to read and write orientation and similarity metric data related to the experimental EBSD data. The orientations are stored in orix Rotation objects which uses NumPy internally. The arrays can easily be extracted prior to any computation allowing the algorithms to avoid pure Python objects, before creating new refined Rotation objects. Additionally, orix provides a sampling of the RFZ used for the creation of the dictionary used in DI. In this work orix 0.5.1 was used.

3.2 kikuchipy

The refinement algorithms are built as an extension to kikuchipy [3], and expects signal classes from kikuchipy as input. This is the case for the master patterns used in the simulation of EBSD patterns in the refinement process, and for the experimental patterns the simulated patterns are compared up against. The different signal classes inherit from HyperSpy, an open-source Python library for multi-dimensional data analysis [36]. In this work kikuchipy 0.4.dev0 was used.

3.3 EMsoft

EMsoft is a collection of open-source programs for simulation and analysis of various electron microscopy techniques [15]. The master patterns used in both the

initial DI step and the following refinement were simulated in EMsoft 5.0. Furthermore, EMsoft 4.2 was used in Chapter 5 to sample the RFZ, and in Chapter 6 to calculate the misorientation angle in degrees.

Chapter 4

Algorithms

The refinement of resulting orientations and projection center estimates from EBSD indexing is an optimization problem consisting of n smaller optimization problems, where n is equal to the number of experimental patterns. As each of the sub-problems are independent of each other, the task is *embarrassingly parallel* [37]. Figure 4.1 shows how an example task graph for the refinement of three patterns would look like, where it is clear that each of the refinement steps do not need to communicate with each other. The refinement process of a single pattern is roughly described by the flowchart in Figure 4.2.

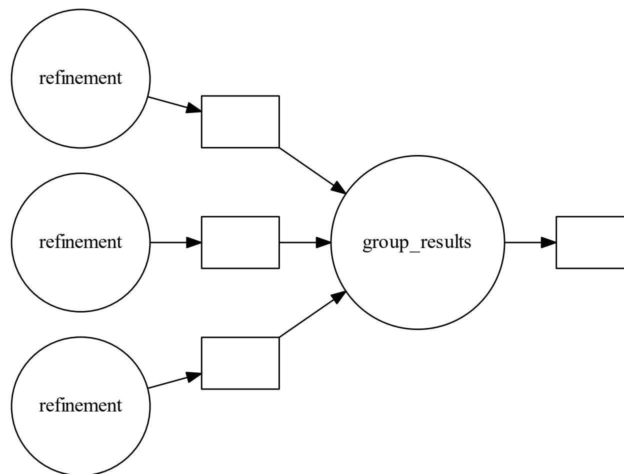


Figure 4.1: Example task graph of the refinement process of three experimental patterns. The refinement of each pattern does not need to communicate with each other, highlighting the embarrassingly parallel nature of the optimization problem.

The implementation in this work consists of three different methods. The

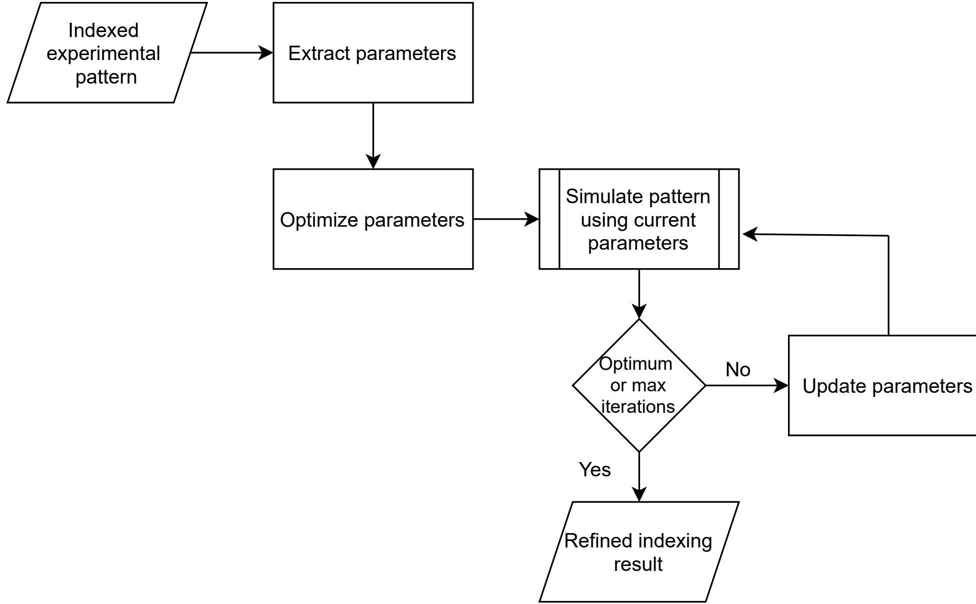


Figure 4.2: Description of single pattern refinement, where the parameters that are optimized could be the Bunge-Euler angles, projection center coordinates, or both.

methods either refine the resulting orientations from any sort of indexing routine, individual projection centers with specified orientations, or both at the same time. The methods are structured similarly, and for every experimental pattern and set of parameters, a wrapper function is called to prepare the parameters before being directed to the user specified SciPy optimization method. The optimization method takes in an objective function, that depends on the parameters being refined. All the supported optimization methods are minimization methods attempting to minimize the objective function value. The objective functions simulate a single EBSD pattern, as described in Section 2.2.2, and uses the normalized cross-correlation coefficient (NCC)

$$r = \frac{\sum_1^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_1^n (x_i - \bar{x})^2 \cdot \sum_1^n (y_i - \bar{y})^2}} \tag{4.1}$$

where x_i and y_i represents the individual pixel values, with $\bar{x}$ and $\bar{y}$ being the mean pixel value across the patterns for the experimental and simulated patterns, respectively, to calculate the similarities between the patterns. If the two patterns were identical the NCC value would be $r = 1$, and lower similarity would result in a smaller r. The objective function attempts to maximize the r-value, and the best parameter values are stored and after every experimental pattern has been refined the updated parameters are returned to the user.

Until the refinement process is merged into kikuchipy, the latest implementation of the algorithms can be found on GitHub [38]. The source code is also in Appendix A.

Chapter 5

Experimental

All the experimental data presented in this work has been indexed, prior to refinement, in the way described in the kikuchipy user guides [3] with a few minor modifications described in the following sections. The dictionary indexing procedure in kikuchipy starts with the loading of experimental data, followed by a pattern processing step where the static and dynamic backgrounds of the experimental patterns are removed. An EMsoft master pattern is then loaded, and a dictionary is created by simulating a set of orientations. The orientations are a result of a uniform sampling of the RFZ with a specified characteristic length. The characteristic length of the sampling with orix was set to 1.4°. Once the dictionary is in place we match every pattern in the dictionary with every pattern in the experimental data set and save the best orientation per scan point. The inverse pole figure (IPFs) maps presented in Chapter 6 are colored according to the color key in figure 5.1 created in MTEX [39].

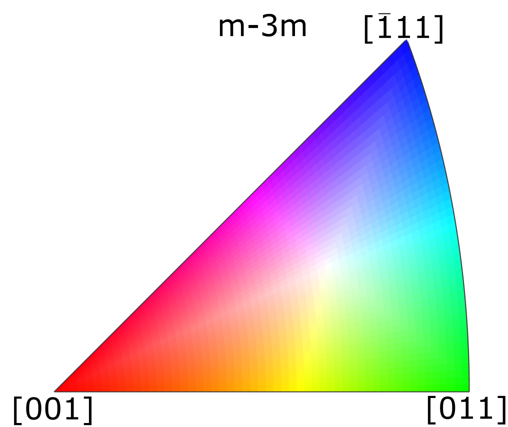


Figure 5.1: m-3m IPF color key.

5.1 Refinement Performance Metrics

In this section various tests are described which evaluates the performance of the refinement algorithms. Both, the actual refinement results and the behaviour of the code when given more resources are evaluated.

5.1.1 Testing of Various Optimization Algorithms

The refinement algorithms support the local optimizers: Nelder-Mead and the modified Powell's algorithm, and the global optimizers: basin-hopping, dual annealing, and differential evolution. To test the different algorithms and their effectiveness, the orientation refinement algorithm was run on the small experimental Nickel test data set included in kikuchipy. The data set is cut-out of scan 1 in the data set presented in Section 5.2. The dictionary used in the pattern matching was created using the EMsoft sampling routine with $N = 100$ resulting in a dictionary of 333 227 patterns.

The comparison was tested using the default SciPy parameters. Basin-hopping and dual annealing both used Nelder-Mead as their local minimizers. Differential evolution and dual annealing had their bounds set to $\pm 1°$ for the Bunge-Euler angles (ϕ_1, Φ, ϕ_2).

5.1.2 Effects of Inaccurate Projection Center Estimates

To test the limits of the refinement algorithms, due to inaccurate projection center estimates, inspiration was taken from Singh *et al.* [9] and 1000 uniformly distributed orientations were sampled using orix, and a simulated experimental data set was created using kikuchipy from those orientations. The ground truth projection center (x^*, y^*, z^*), in TSL notation, was set to $(0.5070, 0.7230, 0.5613)$, with the inaccurate projection center, (x_i^*, y_i^*, z_i^*), defined as:

$$x_i^* = x^* \cdot (1 + \frac{\epsilon}{100}),$$

$$y_i^* = y^* \cdot (1 + \frac{\epsilon}{100}),$$

$$z_i^* = z^* \cdot (1 + \frac{\epsilon}{100}),$$

where $\epsilon \in [-7, 7]$, for a total of $\pm 7\%$ error in the projection center parameters.

For each inaccurate set of projection center estimates, a dictionary was first sampled using orix, before being generated and matched to the simulated experimental data in kikuchipy. Orientation solutions from the pattern matching routine were then refined using the algorithms presented in this work with the Nelder-Mead method as the optimizer of choice.

5.1.3 Effect of Improved Resources

To evaluate how the implementation behaves given more resources, Code listing B.1, which attempts to time the orientation refinement of 10 000 (100 × 100) EBSD patterns indexed using a characteristic length of 4°, was run on the NTNU IDUN computing cluster [4] with varying number of available cores.

5.2 Nickel Acquired with Increasing Camera Gain

To test the effect of increasing noise on the orientation refinement, scan 1, scan 6, and scan 10 were taken from Ånes *et al.* [40], in which ten nickel data sets were collected consecutively, from the same region of interest, with increasing camera gain. The data was obtained with a NORDIF UF-1100 camera on a Hitachi SU-6600 FEG SEM [10]. The general acquisition parameters are presented in Table 5.1, with the scan specific parameters in Table 5.2.

Table 5.1: Acquisition parameters for the nickel data set, with the projection center, (x^*, y^*, z^*), in TSL convention [10].

Voltage	20 kV	(x^*, y^*, z^*)	(0.4210, 0.7794, 0.5049)
Scan size $(s_x \times s_y)$	(300×223.5) μm^2	Scan step size	1.5 μm
Detector size $(d_x \times d_y)$	(480 × 480)	Binning	8

Table 5.2: Camera gain and exposure time, t, for the different nickel scans. Adapted from [10].

Scan	1	6	10
Gain [dB]	0	15	24
t [ms]	3.50	0.65	0.25

The nickel master pattern, shown in Figure 5.2 was simulated in EMsoft 5.0 using the crystal data in Table 5.3 and the Monte Carlo and master pattern parameters in Table 5.4. Prior to indexing, the patterns in each scan had their static and dynamic backgrounds removed. Additionally, in order to increase the signal-to-noise ratio in the experimental patterns further the patterns were averaged with their nearest neighbors using a (3 × 3) Gaussian window, the interested reader is encouraged to visit the kikuchipy pattern processing user guide for more details.

5.3 Simulated Large Scan of Nickel

The DI implementation in kikuchipy assumes a single, fixed projection center to keep the dictionary a manageable size. This assumption is only valid for small

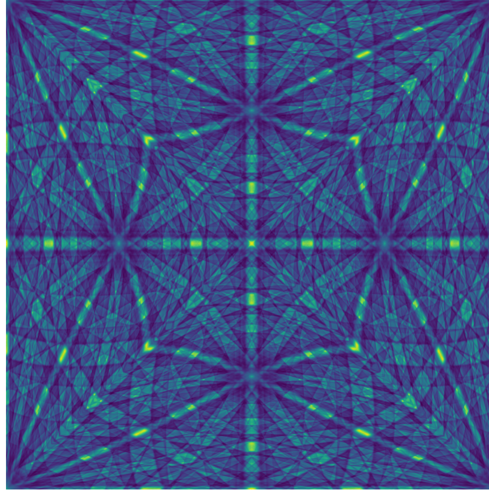


Figure 5.2: Nickel master pattern at 20 keV, in the Roşca-Lambert projection simulated in EMsoft 5.0

Table 5.3: Crystal data parameters used in the DI routine.

Parameter	Ni	Si
Crystal system	Cubic	Cubic
Space group	225	227
a [nm]	0.35236	0.54307
(α, β, γ)	(90° , 90° , 90°)	(90° , 90° , 90°)
Fractional Coordinates	(0, 0, 0)	(0, 0, 0)
Site occuptation	1	1
Debye-Waller factor [nm^2]	0.0035	0.005

scans where the projection center shift is negligible [9]. It has been shown by Singh *et al.* [9] that the projection center shift can be compensated by an equivalent rotation. The orientation refinement algorithm is tested on a simulated, single crystal, large scan of nickel with the parameters given in Table 5.5. The projection center shift is assumed to follow the model given in Singh *et al.* [9] so that:

$$x_{PC}^{new} = x_{PC} - \Delta x_{TD} \quad (5.1)$$

$$y_{PC}^{new} = y_{PC} - \Delta x_{RD} \cos \alpha \quad (5.2)$$

$$L^{new} = L - \Delta x_{RD} \sin \alpha \quad (5.3)$$

where (x_{PC}, y_{PC}, L) is the projection center in EMsoft convention, $\alpha = \frac{\pi}{2} - \sigma + \theta_c$, where σ is the sample tilt and θ_c is the camera elevation. Δx_{TD} and Δx_{RD} represent the step size in the TD and RD direction, respectively. Figure 5.3 shows

Table 5.4: Monte Carlo and master pattern simulation parameters.

Parameter	
Mode	Full
Sample tilt from horizontal	70°
Sample tilt around RD axis	0°
Pixels along x-direction of square projection	501
Number of incident electrons	2e9
Multiplier	1
Incident beam energy	20 keV
Minimum energy to consider	5 keV
Energy bin size	1 keV
Maximum depth to consider for exit depth statistics	100 nm
Depth step size	1 nm
Strong beam cutoff	4
Weak beam cutoff	8
Complete cutoff	50
Maximum excitation error to include	1 nm^{-1}
Smallest d-spacing to take into account	0.05 nm
NUmber of pixels along x-direction of the square master pattern	500

the simulated pattern at the center of the scan with the projection center marked in yellow. The projection center for this pattern is also the one that is used during the DI. Figure 5.4 shows the simulated patterns at the corner of the simulated large scan, with the original projection center marked in yellow. The patterns show a large projection center shift, and thus the assumption of a constant projection center is not valid for scans of this size. The master pattern used during the DI is the same as in Section 5.2

Singh *et al.* [9] proposed an orientation and projection center correction step after the initial DI and orientation refinement, followed by a final orientation refinement. Code listing B.5 gives a Python method to calculate the proposed orientation correction. In this work the initial and final refinement steps of the *three-step* refinement were done using the Nelder-Mead algorithm with the default parameters.

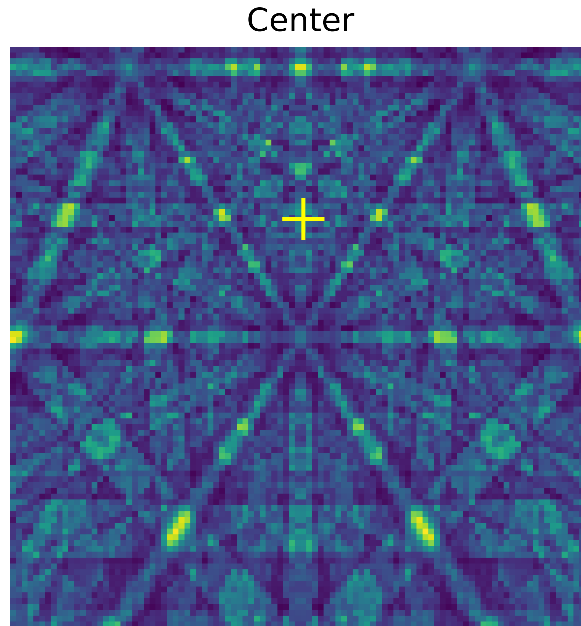


Figure 5.3: Simulated nickel pattern at the center of the simulated large scan, with the projection center marked with a yellow marker.

Table 5.5: Parameters for simulated, single crystal, large scan of nickel. Projection center, (x^*, y^*, z^*), is given in TSL convention.

Voltage	20 kV	(x^*, y^*, z^*)	(0.5, 0.7083, 0.4464)
Scan size $(s_x \times s_y)$	$(2000 \times 3000)\ \mu m^2$	Scan step size	20 μm
Detector size $(d_x \times d_y)$	(480×480)	Binning	4.8
Pixel size	70 μm	Sample tilt	70°
Detector tilt	0°	Bunge-Euler angles	(270° , 45° , 45°)

5.4 Single Crystal Silicon Wafer

The *three-step* refinement was also tested on a real, large, single crystal silicon scan. The patterns were acquired by a NORDIF UF-420 detector on a Zeiss Supra 55VP FEG SEM. The acquisition parameters are presented in Table 5.5, where the projection center was estimated in kikuchipy using the *moving-screen* technique from Hjelen *et al.* [41]. The silicon master pattern used under DI was simulated in EMsoft 5.0 using the crystal data in Table 5.3 and the Monte Carlo and master pattern parameters in Table 5.4.

Figure 5.5 shows an experimental EBSD pattern from the center of the large scan before and after processing in kikuchipy. The Kikuchi bands are now more defined, however, due to the circular nature of the detector the areas outside the scope of the detector do not contain any useful information, while they are still part of the square pattern used in the pattern matching routine. This might lead to

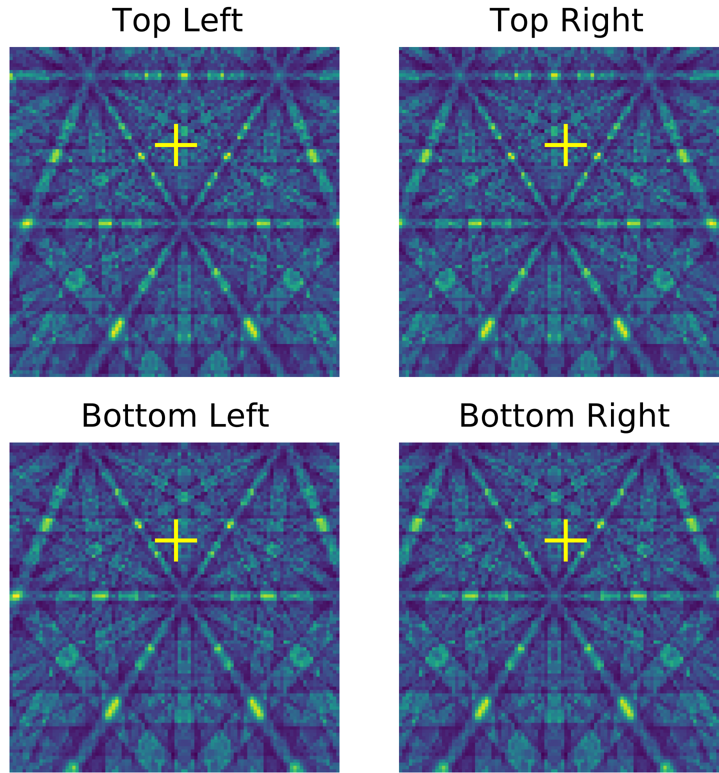


Figure 5.4: Simulated nickel patterns at the corners of the simulated large scan, with the original projection center marked with a yellow marker.

misindexing, and as an attempted workaround scikit-image's triangular threshold filter [42] was used to create a mask that could be applied to the experimental and simulated patterns. The code for the mask creation is available in Code listing B.6. Figure 5.6 shows the transformation of the central experimental pattern and the simulated pattern for the (0°, 0°, 0°) orientation.

Table 5.6: Acquisition parameters for the single crystal silicon data set, with the projection center, (x^*, y^*, z^*), in TSL convention.

Voltage	20 kV	(x^*, y^*, z^*)	(0.5123, 0.8606, 0.4981)
Scan size ($s_x \times s_y$)	$(1960 \times 2000)\mu m^2$	Scan step size	40 μm
Detector size ($d_x \times d_y$)	(480 × 480)	Binning	1
Sample tilt	70°	Detector tilt	0°

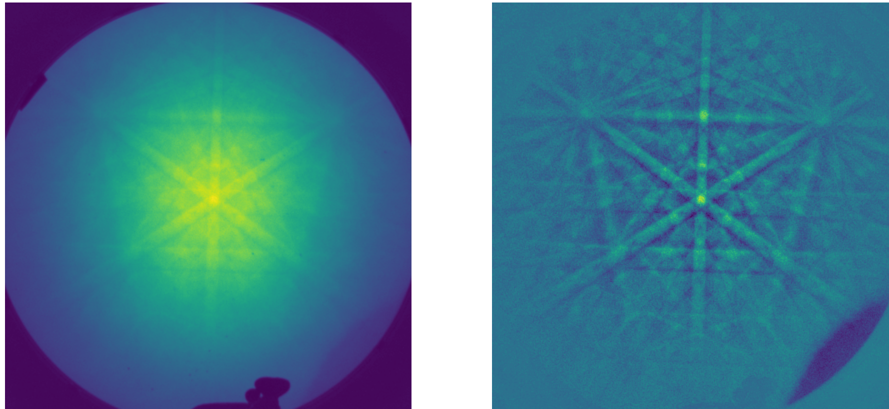


Figure 5.5: EBSD pattern at location 23, 24 in the Si scan before and after pattern processing in kikuchipy.

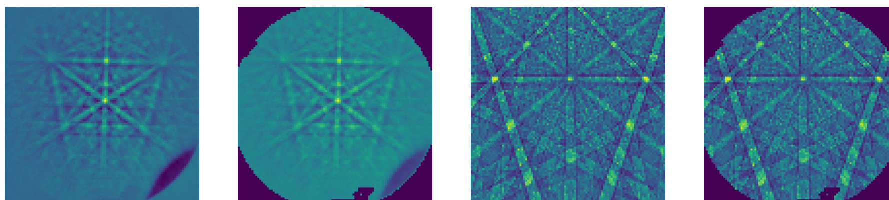


Figure 5.6: Experimental and simulated EBSD patterns from the Si scan after they have had a mask applied to them.

Chapter 6

Results and Discussion

In this chapter the performance of the refinement algorithms are evaluated in different ways. These include the how the algorithms react to being given more resources to work with, the effects of inaccurate simulation parameters, and refining orientations from dictionary indexed data sets with vastly different size properties.

6.1 Refinement Performance Metrics

6.1.1 Testing of Various Optimization Algorithms

Table 6.1 shows the performance of the different optimizers. As one would expect the local optimizers finish much more quickly than the global optimizers as they have a narrower search space. What is not expected is that the local optimizers perform just as well as the global ones. The reasoning behind this is that the the default SciPy parameters are not well suited for this data set, making the global optimizers perform suboptimally. In order for the global optimizers to perform as expected, effort needs to be put in to ensure that the optimization parameters are suited for the data set in question. This was not the focus of this work, so just the default parameters were tested. However, the current implementation of the refinement algorithms support this fine tuning for almost every parameter. Another reason for the good performance of the local optimizers can be attributed to the fact that the dictionary indexing routine contributes with very reasonable initial guesses for the local search.

6.1.2 Effects of Inaccurate Projection Center Estimates

Figure 6.1 shows the mean score against percent error in the projection center parameters. The mean score is defined as the mean of the NCC values assigned to the 1000 simulated patterns. The need for a refinement step after DI as discussed in Singh *et al.* [9] becomes clear as even for completely accurate projection centers the pattern matching routine is unable to get very accurate orientations for

Table 6.1: Performance of various optimization algorithms, on an Intel i7-4790 CPU @ 3.60 GHz, attempting to refine the built in nickel data set in kikuchipy.

Method	Runtime [s]	Mean Score	Min Score	Max Score
DI	104	0.39671	0.13579	0.48152
Nelder-Mead	57	0.44130	0.15170	0.52853
Powell	176	0.44122	0.15169	0.52853
Basin-hopping	8317	0.44144	0.15170	0.52853
Dual Annealing	5774	0.44072	0.15073	0.52853
Differential Evolution	376	0.44040	0.14956	0.52824

pristine patterns. The orientation refinement algorithm presented in this work is able to get very high accurate orientations when the projection center estimate is accurate. However, once the error increases, the benefit of an orientation refinement step falls off quickly and is only marginally better than just DI alone. The full refinement algorithm presented in this work attempts to refine both orientations and projection centers simultaneously, at the cost of longer computation time. The full refinement algorithm using Nelder-Mead is able to get near perfect accurate orientations and projection centers for errors up to 5%. Due to Nelder-Mead being a local optimizer the full refinement completely collapses for larger errors. It should be noted that the simulated data is as good as a match one can get for the pattern matching routine so the realistic error for when the refinement breaks down, is probably lower.

6.1.3 Effect of Improved Resources

The effect on runtime by increasing the number of available CPU cores is shown in Figure 6.2. From the figure it is evident that increasing the number of available CPU cores for the static workload in this evaluation helps increase performance up to a certain point. This proves that there is some degree of parallelism in the implementation of the refinement algorithms presented in this work. Furthermore, from Amdahl's law [37] we have:

$$S = \frac{1}{1 - p + \frac{p}{n}} \tag{6.1}$$

where S is the maximum speedup, for n cores and p is the fraction of the computation that can be done in parallel. In Figure 6.3 the achieved speedup of the orientation refinement is plotted with different p-values. From the figure it is clear that the implementation has a high fraction of parallel computations, and that the benefit of increasing the numbers of core stops at around 10 cores for the workload used in B.1.

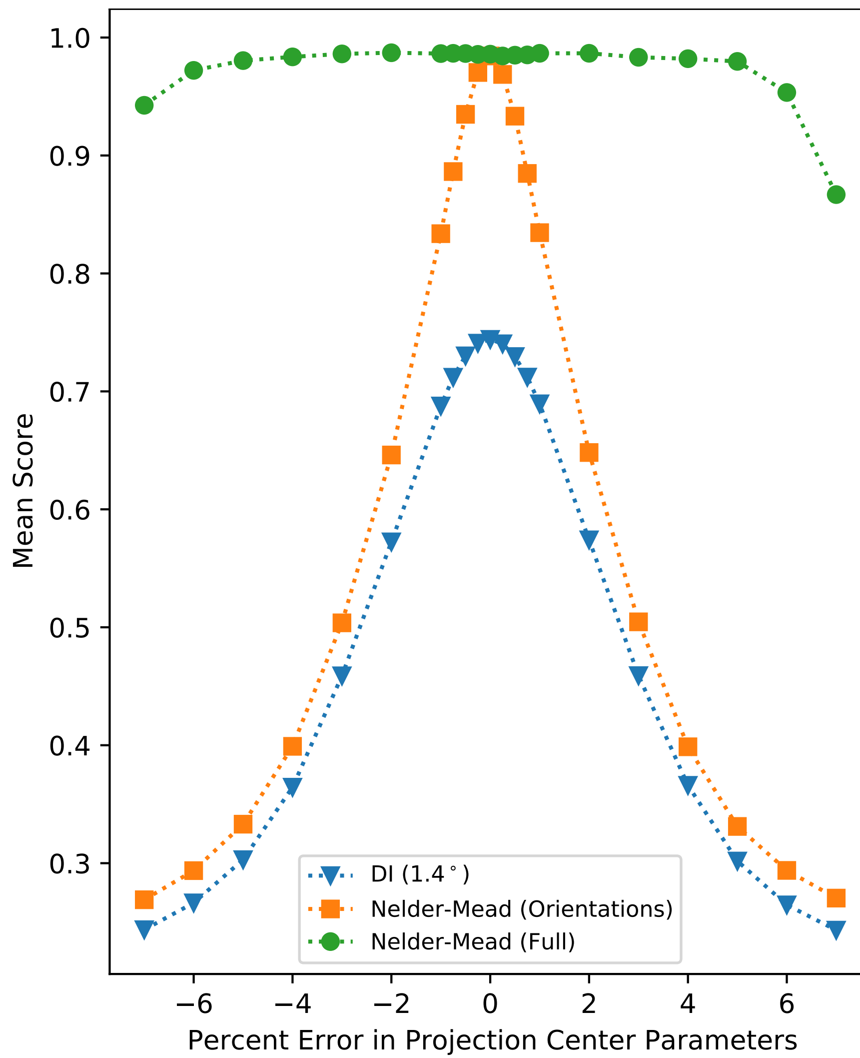


Figure 6.1: Mean NCC-score after refinement of DI results with varying percent error in the projection center parameters used during indexing.

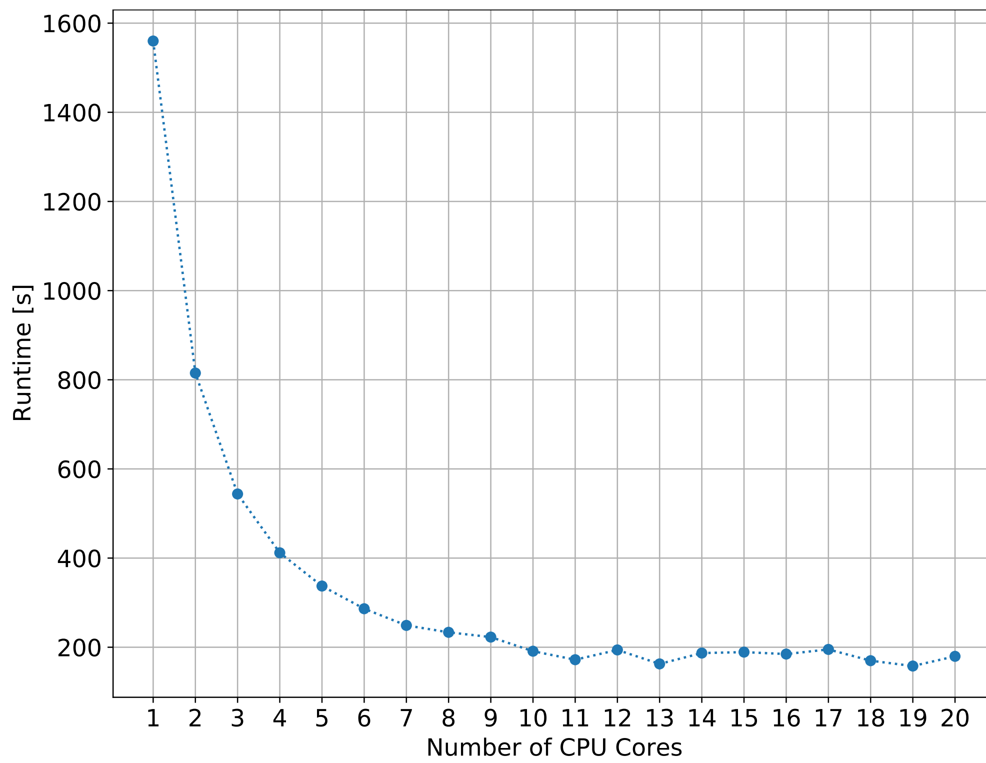


Figure 6.2: Plot of the change in runtime from running the orientation refinement on the NTNU IDUN [4] cluster with increasing number of CPU cores.

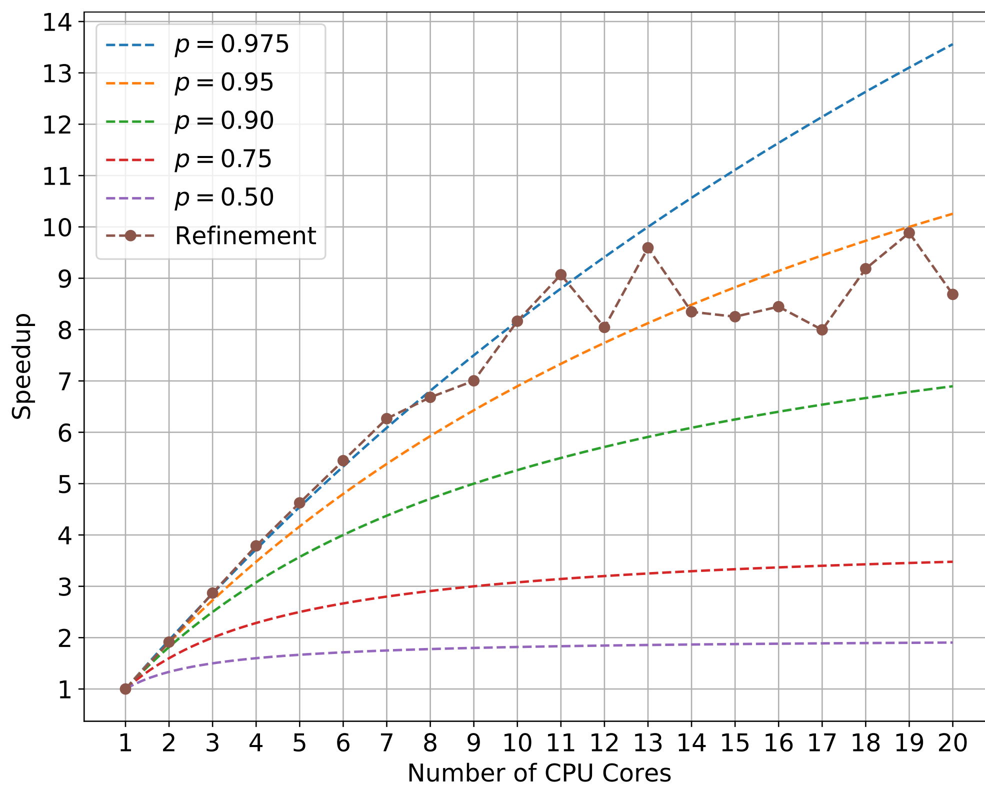


Figure 6.3: Plot of Amdahl's law for different p-values, together with the speedup achieved from running the orientation refinement on the NTNU IDUN [4] cluster with increasing number of CPU cores.

6.2 Nickel Acquired with Increasing Camera Gain

The effects of the pattern processing steps prior to indexing are shown in Figure 6.4, where the first row contains the acquired EBSD patterns and the second row contains the same patterns, but after the pattern processing. It is clear that the pattern processing has increased the signal-to-noise ratio and made the Kikuchi band features more distinct.

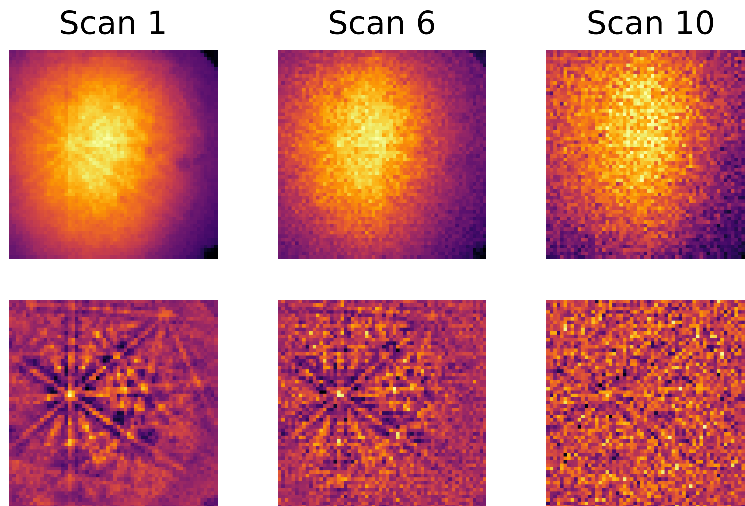


Figure 6.4: Effects of removing the static and dynamic background, and averaging each pattern with its nearest neighbors using a (3×3) Gaussian window in kikuchipy for different scans with increasing gain, at the same location.

Figure 6.5 shows the IPFs after DI in kikuchipy with orix sampling. The robustness of DI to noise is highlighted, as the camera gain increases towards the right the IPF of the different scans stay more or less the same. Figure 6.6 shows the IPFs after being refined with the orientation refinement algorithm presented in this work. The refinement does not drastically alter the orientations indexed by DI. However, it does smooth out the grain orientations.

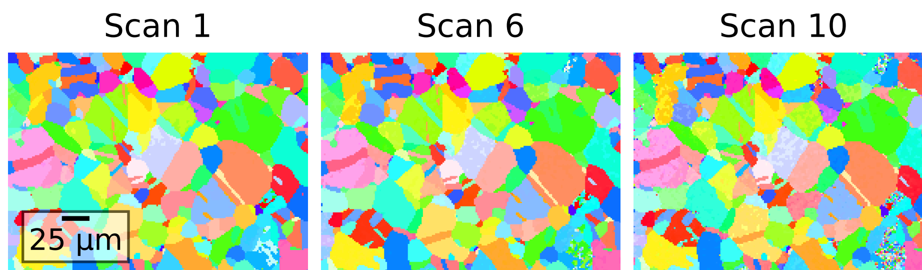


Figure 6.5: IPFs of kikuchipy DI results using orix sampling.

To better see the effect of the refinement, the Gaussian kernel estimation, with a smoothing factor of 0.2, of the PDF of the NCC-scores for each of the scans before

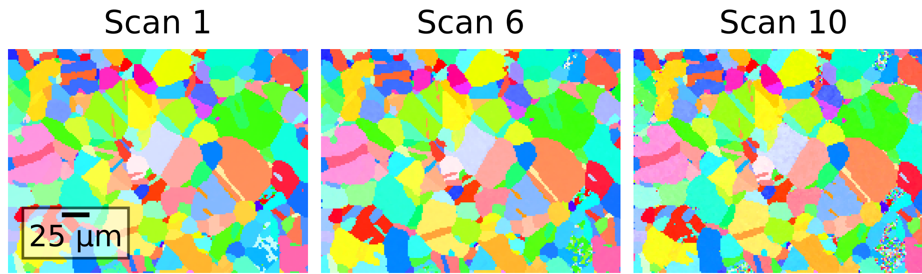


Figure 6.6: IPFs of refined kikuchipy DI results using orix sampling.

and after refinement is shown in Figure 6.7. The PDF is clearly shifted towards the right after refinement, as expected. The distance of the shift is not equal for all the scans, with scan 1 clearly having the larger shift. This is caused by the lower similarity between the simulated and experimental patterns as more noise is introduced into the experimental data.

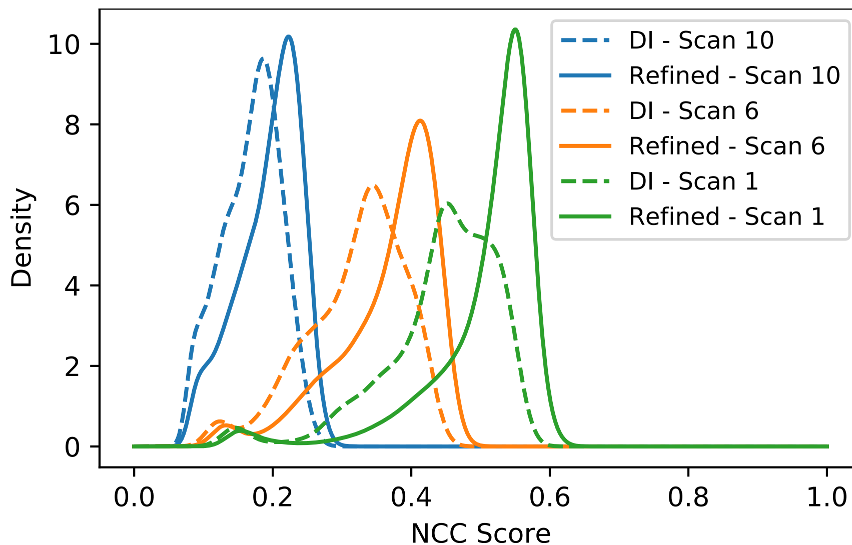


Figure 6.7: Gaussian kernel estimation, with a smoothing factor of 0.2, of the PDF to the NCC-scores in the different scans with orix sampling.

Figure 6.7 displays a small peak towards the low end of the NCC scale for the PDFs of scan 1 and scan 6, whose mean NCC values are much higher. It is suspected that certain areas of the two scans must be wrongly indexed. An easy way to check is to plot the IPFs again, but now set the alpha of each pixel based on the corresponding NCC value. The lowest value would be completely dark, with the best value being transparent. 6.8 shows the IPF of scan 1 with the scores added on top. Clearly there are grains, highlighted in red circles, where the DI routine has been unable to properly match from the dictionary to the experimental patterns as these grains are completely dark.

To check if the problem lies in the kikuchipy DI routine, the scans were indexed

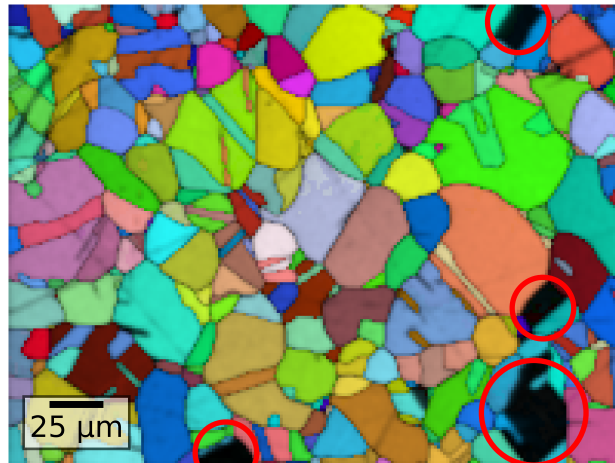


Figure 6.8: Scan 1 after DI in kikuchipy with orix sampling, where the NCC scores are added on top of the image. The red circles highlight areas with very low NCC scores, indicating poor matches.

again but now the dictionary was simulated using orientations from the sampling routine in EMsoft with $N = 100$. Figure 6.9 shows the IPFs after DI in kikuchipy with EMsoft sampling. While the majority of the IPFs are similar to the IPFs in Figure 6.5, there are now red colored grains, which were not present when orix sampling was used.

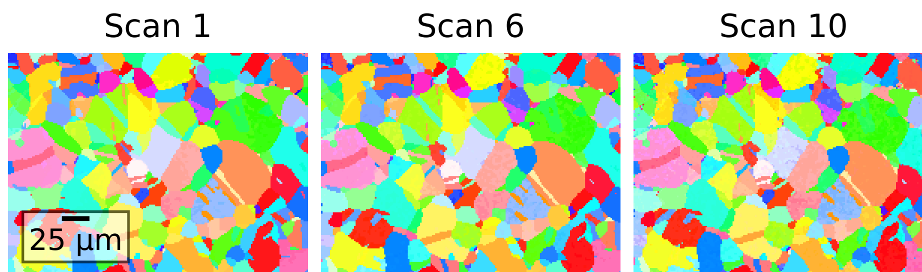


Figure 6.9: IPFs of kikuchipy DI results using EMsoft sampling.

Figure 6.10 shows the difference in NCC scores in scan 1 between DI using EMsoft sampling and DI using orix sampling, with the red circles highlighting the same positions as in Figure 6.8. The large NCC score difference, together with presence of red colored grains is a strong indication that the orix sampling is not good enough for orientations with <100>-normals parallel to the out-of-plane direction].

Figure 6.11 shows the IPFs of the refined kikuchipy DI results with EMsoft sampling. And, just as with orix sampling, there are no major changes between the final orientations. Figure 6.12 shows the Gaussian kernel estimation, with a smoothing factor of 0.2, for the PDFs of the NCC-scores for the different scans. The same properties are observed as for the orix sampling, with one exception

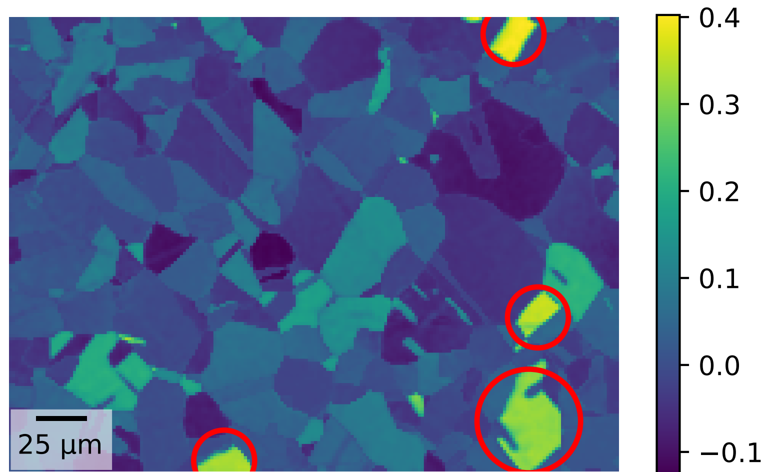


Figure 6.10: NCC-score difference for DI results for scan 1, using EMsoft sampling and orix sampling.

being that the small peaks for scan 1 and scan 6 have no disappeared due to the improved sampling routine.

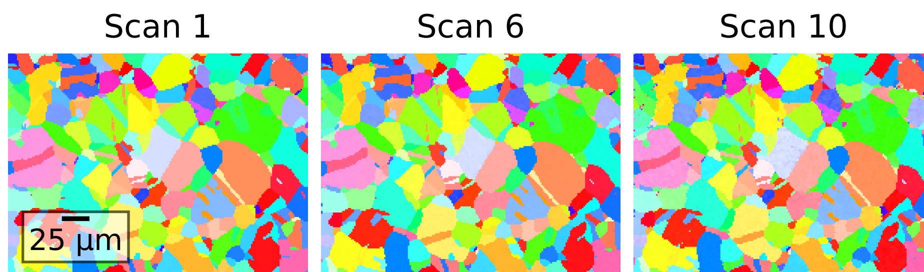


Figure 6.11: IPFs of refined kikuchipy DI results using EMsoft sampling.

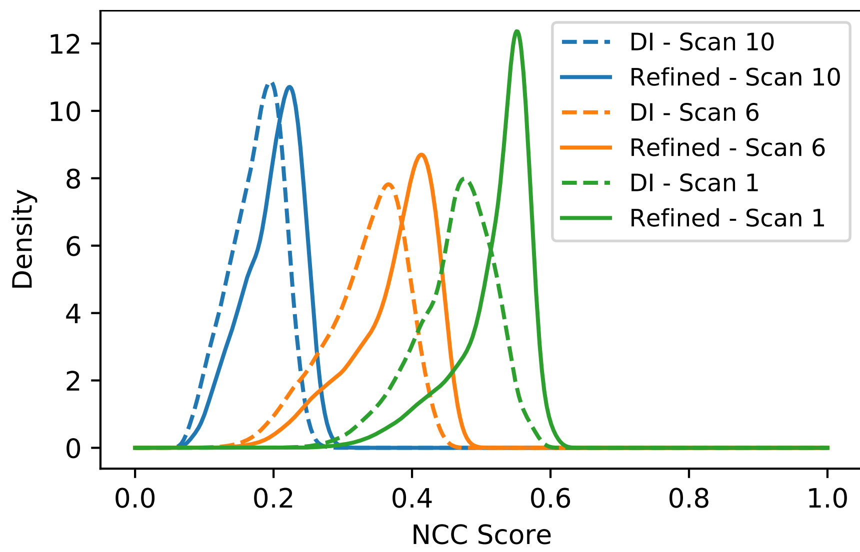


Figure 6.12: Gaussian kernel estimation, with a smoothing factor of 0.2, of the probability density function to the NCC-scores in the different scans with EMsoft sampling

6.3 Simulated Large Scan Ni

Figure 6.13 shows the misorientation angle in degrees for each pixel with the central pixel. The different maps are for the different steps of the *three-step* refinement. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step from Singh *et al.* [9], and D) final refinement using Nelder-Mead.

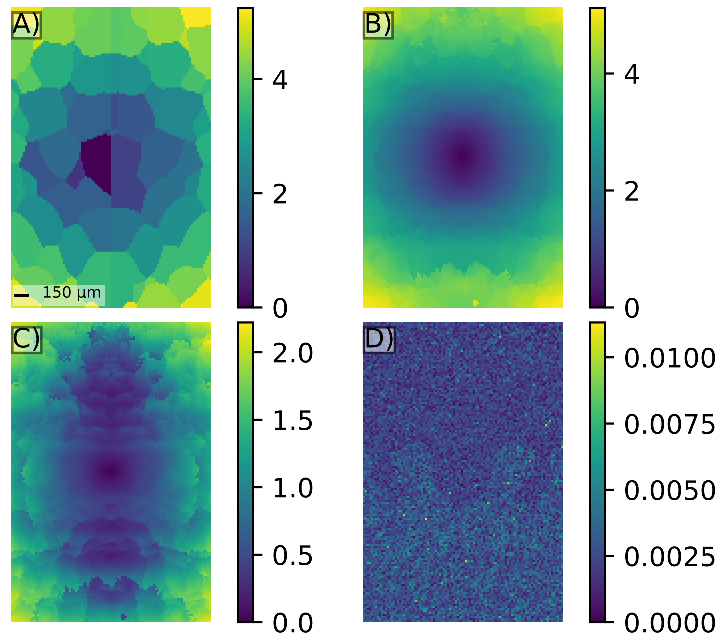


Figure 6.13: Misorientation angle in degrees, to the central pixel in the simulated large Ni scan. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead.

It seems that for this simulated large scan the *three-step* refinement process works really well. However, we note that this is an extreme case where the patterns to be indexed are as close as possible to the dictionary as they are created from the same source. Furthermore, the projection center model is completely accurate as the simulated data set is built from it, which would not be the case in reality.

6.4 Single Crystal Silicon Wafer

The following section is for the large single crystal silicon wafer scan presented in Section 5.4. Figure 6.14 shows the misorientation angle in degrees for each pixel with the central pixel at the different stages of the *three-step* refinement. However, the misorientation has been capped at 10°. From the figure it is pretty clear that neither the DI scan nor the refined result is a single crystal while doing the same approach as for the simulated large Ni scan.

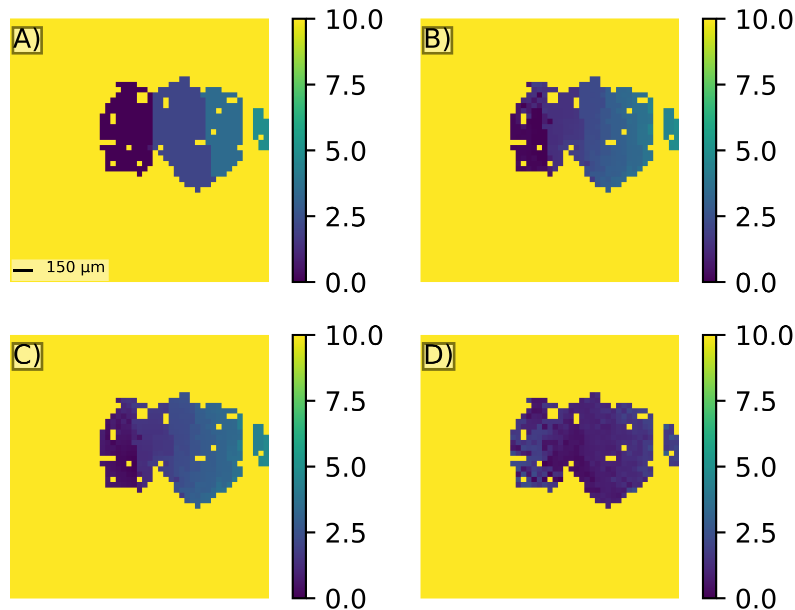


Figure 6.14: Misorientation angle in degrees, capped at 10°, to the central pixel in the Si wafer scan with orix sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead.

The orientation of the single crystal silicon was known to be around (133.3°, 88.7°, 177.8°). To investigate why the previous simulated large Ni scan gave excellent results, while the real large Si scan gave poor results, the method used to simulate the large Ni scan was used to simulate a large Si scan with an orientation set to (133.3°, 88.7°, 177.8°).

6.4.1 Simulated Large Scan Si

Figure 6.15 shows the misorientation angle in degrees for each pixel with the central pixel at the different stages of the *three-step* refinement. Again the maximum misorientation has been set to 10° so that it is easier to see smaller variations inside the scan. The orientation (133.3°, 88.7°, 177.8°) has <100>-normal close to parallel with the out-of-plane direction, so the same *three-step* refinement was tested using sampling from EMsoft with $N = 100$.

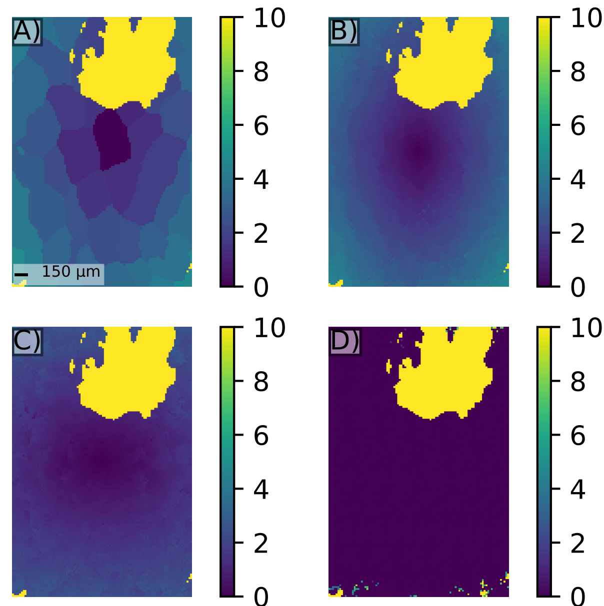


Figure 6.15: Misorientation angle in degrees, capped at 10°, to the central pixel in the simulated large Si scan with orix sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead.

Figure 6.16 shows the misorientation angle in degrees for each pixel with the central pixel at the different stages of the *three-step* refinement, with maximum misorientation capped at 10°. Using the EMsoft sampling the expected behaviour is seen, except for a few points which are clearly misindexed. Again the importance of a good sampling of the RFZ is highlighted and the fact that the sampling provided in orix is not sufficient enough for orientations with <100>-normals parallel to the out-of-plane direction.

6.4.2 Comparison of Orientation Sampling Tools

Based on the results obtained from the simulated large Si scan, the Si wafer was then run through the DI process and the *three-step* refinement using the orientation sampling provided by EMsoft. Figure 6.17 shows the degree of misorientation for each pixel with the central pixel at the different stages of the *three-step* refinement, with a maximum misorientation set to 10°. The process of using EMsoft sampling provides a much better results. However, the result is far from satisfactory as there are large misorientations towards the top corners of the scan. In Figure 6.18 the experimental patterns and their matching simulations from DI are shown. It is evident that at the center, where the projection center estimates are solid, that a good match is found. However, towards the corners DI can no longer guarantee this. The similarity metric used in DI does not account for rotations nor translations, and thus for large scans such as the Si wafer other pattern matching

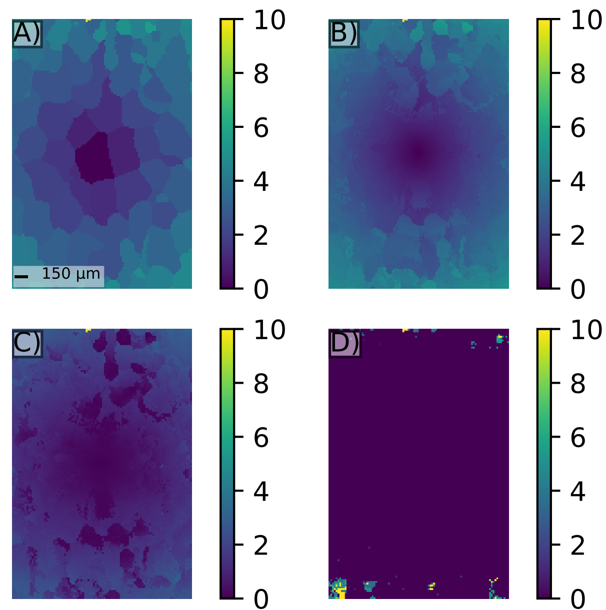


Figure 6.16: Misorientation angle in degrees, capped at 10°, to the central pixel in the simulated large Si scan with EMsoft sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead.

methods, such as Refined Template Matching [43] might be more suited.

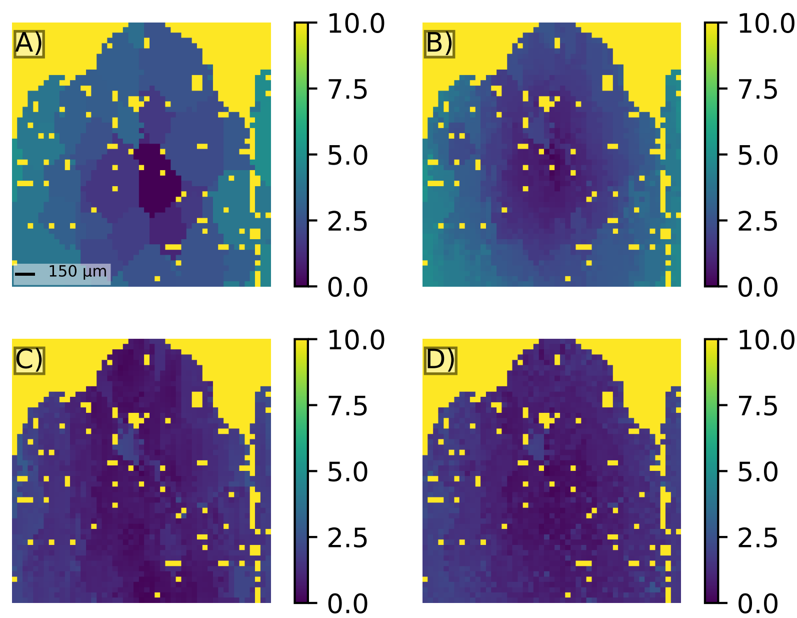


Figure 6.17: Degree of misorientation, capped at 10°, to the central pixel in the Si wafer scan with EMsoft sampling. A) DI, B) orientation refinement using Nelder-Mead, C) orientation and projection center correction step, and D) final orientation refinement using Nelder-Mead.

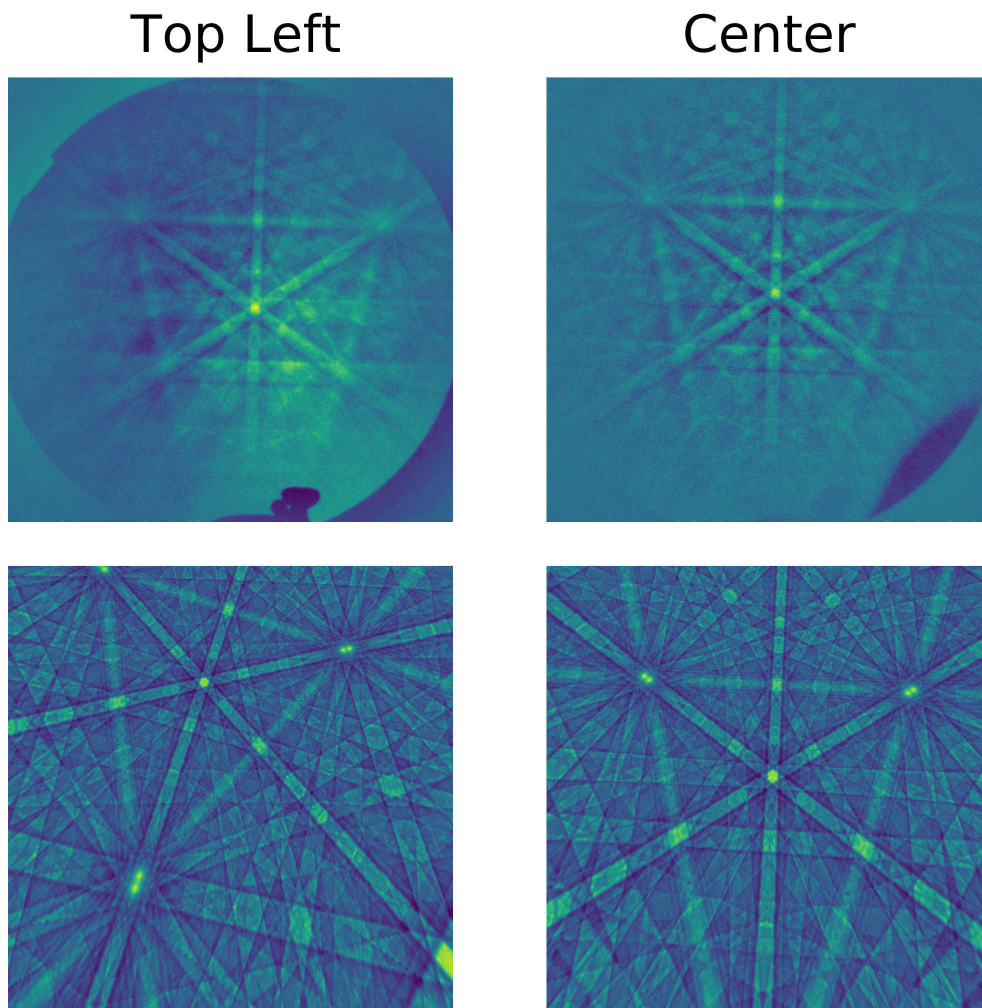


Figure 6.18: The EBSD patterns being used in the DI routine at the top left and center of the scan, compared to the orientation that matched using DI with EMsoft sampling.

Chapter 7

Conclusion

This work has been able to produce a set of algorithms able to refine both the crystal orientations and projection center estimates after EBSD indexing. The implementation is able to run in parallel, with larger-than-memory data sets, both on a normal computer and on a computing cluster. The refinement routines are able to produce reasonable results with just the data to be refined and no experience in optimization is required from the user. Simultaneously, the implementation is flexible enough that if an advanced user wanted to fine-tune the optimization parameters for their specific data set, they could. Additionally, it was concluded that the fundamental sampling method provided by orix is not good enough for orientations with <100>-normals parallel to the out-of-plane direction, and that a proper sampling routine is crucial in order for DI to work. Furthermore, it seems like DI is not a suited routine for indexing experimental scans where the shift in projection center is large.

Chapter 8

Further Work

Going forward the focus should first and foremost be on writing documentation for the implementation, as well as writing unit tests, so that the the refinement feature branch can be merged into the main branch of kikuchipy. Furthermore, it is likely that implementing other pattern matching routines, such as Refined Template Matching, would be beneficial in the long run, and it would be a nice routine to have in the kikuchipy library. Additionally, I would like to see further development of Python equivalent methods of those found in the MTEX Toolbox [39], which does a lot of orientation analysis. Having similar tools available in Python would allow for easier integration of both orientation and pattern analysis.

Bibliography

[1] O. Natlandsmyr, "Algorithm development for matching experimental and simulated electron backscatter diffraction patterns," project thesis, NTNU 2020.

[2] L. Lervik, "Larger than memory dictionaries for EBSD pattern indexing in python," project thesis, NTNU 2020.

[3] H. W. Ånes, O. Natlandsmyr, T. Bergh, and L. Lervik, *Pyxem/kikuchipy: Kikuchipy 0.3.3*, version v0.3.3, Apr. 2021. DOI: 10.5281/zenodo.4707525.

[4] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: 1912.05848 [cs.DC].

[5] A. J. Schwartz, *Electron Backscatter Diffraction in Materials Science*. Springer, 2009, ch. 1.

[6] P. G. Callahan and M. De Graef, "Dynamical electron backscatter diffraction patterns. part i: Pattern simulations," *Microscopy and Microanalysis*, vol. 19, no. 5, pp. 1255–1265, 2013. DOI: 10.1017/S1431927613001840.

[7] A. Winkelmann, C. Trager-Cowan, F. Sweeney, A. P. Day, and P. Parbrook, "Many-beam dynamical simulation of electron backscatter diffraction patterns," *Ultramicroscopy*, vol. 107, no. 4, pp. 414–421, 2007, ISSN: 0304-3991. DOI: https://doi.org/10.1016/j.ultramic.2006.10.006.

[8] K. Lassen, "Automatic high-precision measurements of the location and width of kikuchi bands in electron backscatter diffraction patterns," *Journal of Microscopy*, vol. 190, no. 3, pp. 375–391, 1998. DOI: https://doi.org/10.1046/j.1365-2818.1998.00330.x.

[9] S. Singh, F. Ram, and M. De Graef, "Application of forward models to crystal orientation refinement," *Journal of Applied Crystallography*, vol. 50, no. 6, pp. 1664–1676, Dec. 2017. DOI: 10.1107/S1600576717014200.

[10] H. W. Ånes, J. Hjelen, B. E. Sørensen, A. T. J. van Helvoort, and K. Marthinsen, "Processing and indexing of electron backscatter patterns using open-source software," *IOP Conference Series: Materials Science and Engineering*, vol. 891, p. 012 002, 2020. DOI: 10.1088/1757-899x/891/1/012002.

[11] Y. H. Chen, S. U. Park, D. Wei, G. Newstadt, M. A. Jackson, J. P. Simmons, M. De Graef, and A. O. Hero, "A dictionary approach to electron backscatter diffraction indexing," *Microscopy and Microanalysis*, vol. 21, no. 3, pp. 739–752, 2015. DOI: 10.1017/S1431927615000756.

[12] A. J. Wilkinson, G. Moldovan, T. B. Britton, A. Bewick, R. Clough, and A. I. Kirkland, "Direct detection of electron backscatter diffraction patterns," *Phys. Rev. Lett.*, vol. 111, p. 065 506, 6 Aug. 2013. DOI: 10.1103/PhysRevLett.111.065506.

[13] S. Vespucci, A. Winkelmann, G. Naresh-Kumar, K. P. Mingard, D. Maneuski, P. R. Edwards, A. P. Day, V. O'Shea, and C. Trager-Cowan, "Digital direct electron imaging of energy-filtered electron backscatter diffraction patterns," *Phys. Rev. B*, vol. 92, p. 205 301, 20 Nov. 2015. DOI: 10.1103/PhysRevB.92.205301.

[14] H. W. Ånes, *Figures*, https://github.com/hakonanes/figures, 2020.

[15] S. Singh, F. Ram, and M. De Graef, "Emsoft: Open source software for electron diffraction/image simulations," *Microscopy and Microanalysis*, vol. 23, no. S1, pp. 212–213, 2017. DOI: 10.1017/S143192761700174X.

[16] T. B. Britton, V. S. Tong, J. Hickey, A. Foden, and A. J. Wilkinson, "*AstroEBSD*: exploring new space in pattern indexing with methods launched from an astronomical approach," *Journal of Applied Crystallography*, vol. 51, no. 6, pp. 1525–1534, Dec. 2018. DOI: 10.1107/S1600576718010373.

[17] D. Roşca, "New uniform grids on the sphere," *Astronomy Astrophysics*, vol. 520, A63, 2010, ISSN: 0004-6361. DOI: 10.1051/0004-6361/201015278.

[18] O. Engler and V. Randle, *Introduction to texture analysis*, 2nd ed. CRC Press Inc, 2009, pp. 46–50.

[19] K. Marquardt, M. De Graef, S. Singh, H. Marquardt, A. Rosenthal, and S. Koizuimi, "Quantitative electron backscatter diffraction (ebsd) data analyses using the dictionary indexing (di) approach: Overcoming indexing difficulties on geological materials," *American Mineralogist*, vol. 102, no. 9, pp. 1843–1855, 2017, ISSN: 0003-004X. DOI: 10.2138/am-2017-6062.

[20] A. Winkelmann, *The normalized cross correlation coefficient — xcdskd version v0.1-29-g8cf606c*, Accessed: 2021-06-08. [Online]. Available: https://xcdskd.readthedocs.io/en/latest/cross_correlation/cross_correlation_coefficient.html#Example-Data:-Kikuchi-Pattern-Fits.

[21] I. Simonsen, *TFY4235/FY8904 : Computational physics*, Optimization. Accessed: 2021-06-09, 2018. [Online]. Available: http://web.phys.ntnu.no/~ingves/Teaching/TFY4235/Download/TFY4235_Slides_2018.pdf.

[22] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, Jan. 1965, ISSN: 0010-4620. DOI: 10.1093/comjnl/7.4.308.

[23] F. Gao and L. Han, "Implementing the nelder-mead simplex algorithm with adaptive parameters," *Computational Optimization and Applications*, vol. 51, no. 1, pp. 259–277, 2012, ISSN: 0926-6003. DOI: 10.1007/s10589-010-9329-3.

[24] M. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, pp. 155–162, 1964.

[25] D. J. Wales and J. P. K. Doye, "Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms," *The Journal of Physical Chemistry A*, vol. 101, no. 28, pp. 5111–5116, 1997. DOI: 10.1021/jp970984n.

[26] Y. Xiang, D. Sun, W. Fan, and X. Gong, "Generalized simulated annealing algorithm and its application to the thomson model," *Physics Letters A*, vol. 233, no. 3, pp. 216–220, 1997, ISSN: 0375-9601. DOI: https://doi.org/10.1016/S0375-9601(97)00474-X.

[27] R. Storn and K. Price, "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, pp. 341–359, Jan. 1997. DOI: 10.1023/A:1008202821328.

[28] E. L. Pang, P. M. Larsen, and C. A. Schuh, "Global optimization for accurate determination of ebsd pattern centers," *Ultramicroscopy*, vol. 209, p. 112 876, 2020, ISSN: 0304-3991. DOI: 10.1016/j.ultramic.2019.112876.

[29] *Python package index*, https://pypi.org/, Accessed: 2021-06-11.

[30] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2.

[31] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[32] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org.

[33] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A LLVM-Based python JIT compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15, Austin, Texas: Association for Computing Machinery, 2015, ISBN: 9781450340052. DOI: 10.1145/2833157.2833162.

[34] P. Crout, H. W. Ånes, D. N. Johnstone, B. Martineau, and S. Høgås, “Pyxem/orix: Orix 0.5.1,” Nov. 2020. DOI: 10.5281/zenodo.4233526.

[35] D. N. Johnstone, B. H. Martineau, P. Crout, P. A. Midgley, and A. S. Eggeman, “Density-based clustering of crystal (mis) orientations and the orix Python library,” *Journal of Applied Crystallography*, vol. 53, no. 5, 2020. DOI: 10.1107/S1600576720011103.

[36] F. de la Peña, E. Prestat, V. T. Fauske, P. Burdet, T. Furnival, P. Jokubauskas, M. Nord, T. Ostasevicius, K. E. MacArthur, D. N. Johnstone, and et al., “Hyperspy/hyperspy: Release v1.6.1,” Nov. 2020. DOI: 10.5281/zenodo.4294676.

[37] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, pp. 13–15, ISBN: 9780123973375.

[38] L. Lervik, *Kikuchipy refinement*, 2021 publisher = GitHub, journal = GitHub repository, howpublished = https://github.com/friedkitteh/kikuchipy/blob/refinement/kikuchipy/indexing/refinement.py,

[39] F. Bachmann, R. Hielscher, and H. Schaeben, “Texture analysis with mtex – free and open source software toolbox,” in *Texture and Anisotropy of Polycrystals III*, ser. Solid State Phenomena, vol. 160, Trans Tech Publications Ltd, Mar. 2010, pp. 63–68. DOI: 10.4028/www.scientific.net/SSP.160.63.

[40] H. W. Ånes, J. Hjelen, A. T. J. van Helvoort, and K. Marthinsen, *Electron backscatter patterns from nickel acquired with varying camera gain*, The data was acquired while Håkon Wiik Ånes received financial support from the Norwegian University of Science and Technology (NTNU) through the NTNU Aluminium Product Innovation Centre (NAPIC)., Sep. 2019. DOI: 10.5281/zenodo.3265037.

[41] J. Hjelen, E. Hoel, and R. Ørsund, “Electron diffraction in the sem,” *Micron and Microscopica Acta*, vol. 22, no. 1, pp. 137–138, 1991, ISSN: 0739-6260. DOI: https://doi.org/10.1016/0739-6260(91)90128-M.

[42] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, “Scikit-image: Image processing in Python,” *PeerJ*, vol. 2, e453, Jun. 2014, ISSN: 2167-8359. DOI: 10.7717/peerj.453.

[43] A. Foden, D. Collins, A. Wilkinson, and T. Britton, "Indexing electron backscatter diffraction patterns with a refined template matching approach," *Ultramicroscopy*, vol. 207, p. 112 845, 2019, ISSN: 0304-3991. DOI: https://doi.org/10.1016/j.ultramic.2019.112845.

Appendix A

Refinement Implementation

In this chapter the code used in the refinement process, including optimization code, similarity metrics and single pattern simulation code is presented. As mentioned in Chapter 4, the latest updates and a better overview of the source code can be found on GitHub [38] until it is merged into the master branch of kikuchipy.

Code listing A.1: Required imports for the code listings presented in this chapter.

```
import sys
from typing import, Union


import dask
from dask.diagnostics import ProgressBar
import numba
import numpy as np
from orix.crystal_map import CrystalMap
from orix.quaternion import Rotation
import scipy.optimize
```

A.1 Calling Functions

The calling functions are static methods of the Refinement class.

A.1.1 Full Refinement

Code listing A.2: Full refinement method.

```
@staticmethod
def refine_xmap(
    xmap,
    mp,
    exp,
    det,
    energy,
    mask=1,
    method="minimize",
    method_kwargs=None,
```

```
    trust_region=None,
    compute=True,
):
    if method == "minimize" and not method_kwargs:
        method_kwargs = {"method": "Nelder-Mead"}
    elif not method_kwargs:
        method_kwargs = {}
    method = getattr(scipy.optimize, method)

    # Convert from Quaternions to Euler angles
    with np.errstate(divide="ignore", invalid="ignore"):
        euler = Rotation.to_euler(xmap.rotations)

    # Extract best rotation from xmap if given more than 1
    if len(euler.shape) > 2:
        euler = euler[:, 0, :]

    if not trust_region:
        trust_region = [
            0.0174532925,
            0.0174532925,
            0.0174532925,
            0.05,
            0.05,
            0.05,
        ]
    else:
        trust_region = (
            np.deg2rad(trust_region[:3]).tolist() + trust_region[3:]
        )

    exp.rescale_intensity(dtype_out=np.float32)
    exp_data = exp.data
    exp_shape = exp_data.shape

    pc = det.pc

    # Set the PC equal across the scan if not given
    if len(pc) == 1:
        pc_val = pc[0]
        pc = np.full((exp_shape[0] * exp_shape[1], 3), pc_val)
    # Should raise error here if len pc not equal to scan size

    # 2D nav-dim
    if len(exp_shape) == 4:
        exp_data = exp_data.reshape(
            (exp_shape[0] * exp_shape[1], exp_shape[2], exp_shape[3])
        )
    elif len(exp_shape) == 2:  # 0D nav-dim
        exp_data = exp_data.reshape(((1,) + exp_data.shape))

    (
        master_north,
        master_south,
        npx,
        npy,
        scale,
    ) = _get_single_pattern_params(mp, det, energy)

    theta_c = np.deg2rad(det.tilt)
```

```
    sigma = np.deg2rad(det.sample_tilt)
    alpha = (np.pi / 2) - sigma + theta_c

    detector_data = [det.ncols, det.nrows, det.px_size, alpha]

    pre_args = (
        master_north,
        master_south,
        npx,
        npy,
        scale,
        detector_data,
        mask,
    )

    pre_args = dask.delayed(pre_args)
    trust_region = dask.delayed(trust_region)

    if isinstance(exp_data, dask.array.core.Array):
        patterns_in_chunk = exp_data.chunks[0]
        partitons = exp_data.to_delayed()  # List of delayed objects
        # equal to the number of chunks
        inner_index = 0
        refined_params = []
        for k, part in enumerate(partitons):
            data = part[0, 0]
            num_patterns = patterns_in_chunk[k]
            for i in range(num_patterns):
                res = dask.delayed(_refine_xmap_solver)(
                    euler[i + inner_index],
                    pc[i + inner_index],
                    data[i],
                    pre_args,
                    method,
                    method_kwargs,
                    trust_region,
                )
                refined_params.append(res)

            inner_index += num_patterns  # Increase the index for
            # the next chunk
    else:  # NumPy array
        refined_params = [
            dask.delayed(_refine_xmap_solver)(
                euler[i],
                pc[i],
                exp_data[i],
                pre_args,
                method,
                method_kwargs,
                trust_region,
            )
            for i in range(euler.shape[0])
        ]
    if compute:
        with ProgressBar():
            print(
                f"Refining {xmap.rotations.shape[0]} orientations and "
                f"projection centers:",
                file=sys.stdout,
```

```python
            )
            results = dask.compute(*refined_params, scheduler="threads")
            refined_euler = np.empty((euler.shape[0], 3), dtype=np.float32)
            refined_pc = np.empty((euler.shape[0], 3), dtype=np.float32)
            refined_scores = np.empty((euler.shape[0]), dtype=np.float32)
            for i in range(euler.shape[0]):
                refined_scores[i] = results[i][0]

                refined_euler[i][0] = results[i][1]
                refined_euler[i][1] = results[i][2]
                refined_euler[i][2] = results[i][3]

                refined_pc[i][0] = results[i][4]
                refined_pc[i][1] = results[i][5]
                refined_pc[i][2] = results[i][6]

            new_det = det.deepcopy()
            new_det.pc = refined_pc
            refined_rotations = Rotation.from_euler(refined_euler)
            xmap_dict = xmap.__dict__

            output = CrystalMap(
                rotations=refined_rotations,
                phase_id=xmap_dict["_phase_id"],
                x=xmap_dict["_x"],
                y=xmap_dict["_y"],
                phase_list=xmap_dict["phases"],
                prop={
                    "scores": refined_scores,
                },
                is_in_data=xmap_dict["is_in_data"],
                scan_unit=xmap_dict["scan_unit"],
            )
    else:
        output = dask.delayed(refined_params)
        new_det = -1
    return output, new_det
```

A.1.2 Orientation Refinement

Code listing A.3: Orientation refinement method.

```python
@staticmethod
def refine_orientations(
    xmap,
    mp,
    exp,
    det,
    energy,
    mask=1,
    method="minimize",
    method_kwargs=None,
    trust_region=None,
    compute=True,
):
    if method == "minimize" and not method_kwargs:
        method_kwargs = {"method": "Nelder-Mead"}
    elif not method_kwargs:
```

```
        method_kwargs = {}
    method = getattr(scipy.optimize, method)

    # Convert from Quaternions to Euler angles
    with np.errstate(divide="ignore", invalid="ignore"):
        euler = Rotation.to_euler(xmap.rotations)

    # Extract best rotation from xmap if given more than 1
    if len(euler.shape) > 2:
        euler = euler[:, 0, :]

    exp.rescale_intensity(dtype_out=np.float32)  # Here we are rescaling
    # the input, we should probably not do this! :)
    exp_data = exp.data
    exp_shape = exp_data.shape

    if len(exp_shape) == 4:
        exp_data = exp_data.reshape(
            (exp_shape[0] * exp_shape[1], exp_shape[2], exp_shape[3])
        )
    elif len(exp_shape) == 2:  # 0D nav-dim
        exp_data = exp_data.reshape(((1,) + exp_data.shape))

    if not trust_region:
        trust_region = [0.0174532925, 0.0174532925, 0.0174532925]  # 1 deg
    else:
        trust_region = np.deg2rad(trust_region)

    scan_points = exp_data.shape[0]

    theta_c = np.deg2rad(det.tilt)
    sigma = np.deg2rad(det.sample_tilt)
    alpha = (np.pi / 2) - sigma + theta_c

    dncols = det.ncols
    dnrows = det.nrows
    px_size = det.px_size


    pc_emsoft = det.pc_emsoft()
    if len(pc_emsoft) == 1:
        xpc = np.full(scan_points, pc_emsoft[..., 0])
        ypc = np.full(scan_points, pc_emsoft[..., 1])
        L = np.full(scan_points, pc_emsoft[..., 2])
    else:  # Should raise error here if shape mismatch with exp!!
        xpc = pc_emsoft[..., 0]
        ypc = pc_emsoft[..., 1]
        L = pc_emsoft[..., 2]

    (
        master_north,
        master_south,
        npx,
        npy,
        scale,
    ) = _get_single_pattern_params(mp, det, energy)

    pre_args = (
        master_north,
        master_south,
```

```
        npx,
        npy,
        scale,
        mask,
    )

    pre_args = dask.delayed(pre_args)
    trust_region = dask.delayed(trust_region)

    if isinstance(exp_data, dask.array.core.Array):
        patterns_in_chunk = exp_data.chunks[0]
        partitons = exp_data.to_delayed()  # List of delayed objects
        # equal to the number of chunks
        inner_index = 0
        refined_params = []
        for k, part in enumerate(partitons):
            data = part[0, 0]
            num_patterns = patterns_in_chunk[k]

            dc = dask.delayed(_fast_get_dc_multiple_pc)(
                xpc[inner_index : num_patterns + inner_index],
                ypc[inner_index : num_patterns + inner_index],
                L[inner_index : num_patterns + inner_index],
                num_patterns,
                dncols,
                dnrows,
                px_size,
                alpha,
            )

            for i in range(num_patterns):
                res = dask.delayed(_refine_orientations_solver)(
                    data[i],
                    euler[inner_index + i],
                    dc[i],
                    method,
                    method_kwargs,
                    pre_args,
                    trust_region,
                )
                refined_params.append(res)

            inner_index += num_patterns  # Increase the index for
            # the next chunk

    else:  # numpy array
        dc = _fast_get_dc_multiple_pc(
            xpc, ypc, L, scan_points, dncols, dnrows, px_size, alpha
        )

        refined_params = [
            dask.delayed(_refine_orientations_solver)(
                exp_data[i],
                euler[i],
                dc[i],
                method,
                method_kwargs,
                pre_args,
                trust_region,
            )
```

```
            for i in range(euler.shape[0])
        ]

    if compute:
        with ProgressBar():
            print(
                f"Refining {xmap.rotations.shape[0]} orientations:",
                file=sys.stdout,
            )
            results = dask.compute(*refined_params)
            refined_euler = np.empty(
                (xmap.rotations.shape[0], 3), dtype=np.float32
            )
            refined_scores = np.empty(
                (xmap.rotations.shape[0]), dtype=np.float32
            )
            for i in range(xmap.rotations.shape[0]):
                refined_scores[i] = results[i][0]

                refined_euler[i][0] = results[i][1]
                refined_euler[i][1] = results[i][2]
                refined_euler[i][2] = results[i][3]

            refined_rotations = Rotation.from_euler(refined_euler)

            xmap_dict = xmap.__dict__

            output = CrystalMap(
                rotations=refined_rotations,
                phase_id=xmap_dict["_phase_id"],
                x=xmap_dict["_x"],
                y=xmap_dict["_y"],
                phase_list=xmap_dict["phases"],
                prop={
                    "scores": refined_scores,
                },
                is_in_data=xmap_dict["is_in_data"],
                scan_unit=xmap_dict["scan_unit"],
            )
    else:
        output = dask.delayed(refined_params)
    return output
```

A.1.3 Projection Center Refinement

Code listing A.4: Projection center refinement method.

```
@staticmethod
def refine_projection_center(
    xmap,
    mp,
    exp,
    det,
    energy,
    mask=1,
    method="minimize",
    method_kwargs=None,
    trust_region=None,
```

```
    compute=True,
):
    if method == "minimize" and not method_kwargs:
        method_kwargs = {"method": "Nelder-Mead"}
    elif not method_kwargs:
        method_kwargs = {}
    method = getattr(scipy.optimize, method)

    # Extract best rotation from xmap if given more than 1
    if len(xmap.rotations.shape) > 1:
        r = xmap.rotations[:, 0].data
    else:
        r = xmap.rotations.data

    exp.rescale_intensity(dtype_out=np.float32)
    exp_data = exp.data
    exp_shape = exp_data.shape

    pc = det.pc

    # Set the PC equal across the scan if not given
    if len(pc) == 1:
        pc_val = pc[0]
        pc = np.full((exp_shape[0] * exp_shape[1], 3), pc_val)
    # Should raise error here if len pc not equal to scan size

    # 2D nav-dim
    if len(exp_shape) == 4:
        exp_data = exp_data.reshape(
            (exp_shape[0] * exp_shape[1], exp_shape[2], exp_shape[3])
        )
    elif len(exp_shape) == 2:  # 0D nav-dim
        exp_data = exp_data.reshape(((1,) + exp_data.shape))

    if not trust_region:
        trust_region = [0.05, 0.05, 0.05]

    (
        master_north,
        master_south,
        npx,
        npy,
        scale,
    ) = _get_single_pattern_params(mp, det, energy)

    theta_c = np.deg2rad(det.tilt)
    sigma = np.deg2rad(det.sample_tilt)
    alpha = (np.pi / 2) - sigma + theta_c

    detector_data = [det.ncols, det.nrows, det.px_size, alpha]

    pre_args = (
        master_north,
        master_south,
        npx,
        npy,
        scale,
        detector_data,
        mask,
    )
```

```
    pre_args = dask.delayed(pre_args)
    trust_region = dask.delayed(trust_region)

    if isinstance(exp_data, dask.array.core.Array):
        patterns_in_chunk = exp_data.chunks[0]
        partitons = exp_data.to_delayed()  # List of delayed objects
        # equal to the number of chunks
        inner_index = 0
        refined_params = []
        for k, part in enumerate(partitons):
            data = part[0, 0]
            num_patterns = patterns_in_chunk[k]
            for i in range(num_patterns):
                res = dask.delayed(_refine_pc_solver)(
                    data[i],
                    r[i + inner_index],
                    pc[i + inner_index],
                    method,
                    method_kwargs,
                    pre_args,
                    trust_region,
                )
                refined_params.append(res)

            inner_index += num_patterns  # Increase the index for
            # the next chunk
    else:  # NumPy array
        refined_params = [
            dask.delayed(_refine_pc_solver)(
                exp_data[i],
                r[i],
                pc[i],
                method,
                method_kwargs,
                pre_args,
                trust_region,
            )
            for i in range(xmap.rotations.shape[0])
        ]

    output = refined_params
    if compute:
        with ProgressBar():
            print(
                f"Refining {xmap.rotations.shape[0]} projection centers:",
                file=sys.stdout,
            )
            results = dask.compute(*refined_params)

            refined_pc = np.empty(
                (xmap.rotations.shape[0], 3), dtype=np.float32
            )
            refined_scores = np.empty(
                (xmap.rotations.shape[0]), dtype=np.float32
            )
            for i in range(xmap.rotations.shape[0]):
                refined_scores[i] = results[i][0]

                refined_pc[i][0] = results[i][1]
```

```
                refined_pc[i][1] = results[i][2]
                refined_pc[i][2] = results[i][3]

            new_det = det.deepcopy()
            new_det.pc = refined_pc

            output = (refined_scores, new_det)

    return output
```

A.2 Solver Functions

A.2.1 Full Refinement

Code listing A.5: Full refinement solver method.

```
def _refine_xmap_solver(
    r, pc, exp, pre_args, method, method_kwargs, trust_region
):
    phi1_0 = r[..., 0]
    Phi_0 = r[..., 1]
    phi2_0 = r[..., 2]
    eu_x0 = np.array((phi1_0, Phi_0, phi2_0))

    args = (exp,) + pre_args

    full_x0 = np.concatenate((eu_x0, pc), axis=None)

    if method.__name__ == "minimize":
        soln = method(
            _full_objective_function_euler,
            x0=full_x0,
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "differential_evolution":
        soln = method(
            _full_objective_function_euler,
            bounds=[
                (full_x0[0] - trust_region[0], full_x0[0] + trust_region[0]),
                (full_x0[1] - trust_region[1], full_x0[1] + trust_region[1]),
                (full_x0[2] - trust_region[2], full_x0[2] + trust_region[2]),
                (full_x0[3] - trust_region[3], full_x0[3] + trust_region[3]),
                (full_x0[4] - trust_region[4], full_x0[4] + trust_region[4]),
                (full_x0[5] - trust_region[5], full_x0[5] + trust_region[5]),
            ],
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "dual_annealing":
        soln = method(
            _full_objective_function_euler,
            bounds=[
                (full_x0[0] - trust_region[0], full_x0[0] + trust_region[0]),
                (full_x0[1] - trust_region[1], full_x0[1] + trust_region[1]),
                (full_x0[2] - trust_region[2], full_x0[2] + trust_region[2]),
                (full_x0[3] - trust_region[3], full_x0[3] + trust_region[3]),
```

```
                (full_x0[4] - trust_region[4], full_x0[4] + trust_region[4]),
                (full_x0[5] - trust_region[5], full_x0[5] + trust_region[5]),
            ],
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "basinhopping":
        method_kwargs["minimizer_kwargs"]["args"] = args
        soln = method(
            _full_objective_function_euler,
            x0=full_x0,
            **method_kwargs,
        )

    score = 1 - soln.fun
    phi1 = soln.x[0]
    Phi = soln.x[1]
    phi2 = soln.x[2]
    pcx = soln.x[3]
    pxy = soln.x[4]
    pxz = soln.x[5]

    return (score, phi1, Phi, phi2, pcx, pxy, pxz)
```

A.2.2 Orientation Refinement

Code listing A.6: Orientation refinement solver function.

```
def _refine_orientations_solver(
    exp, r, dc, method, method_kwargs, pre_args, trust_region
):

    phi1 = r[..., 0]
    Phi = r[..., 1]
    phi2 = r[..., 2]

    args = (exp,) + pre_args + (dc,)

    r_x0 = np.array((phi1, Phi, phi2), dtype=np.float32)

    if method.__name__ == "minimize":
        soln = method(
            _orientation_objective_function_euler,
            x0=r_x0,
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "differential_evolution":
        soln = method(
            _orientation_objective_function_euler,
            bounds=[
                (r_x0[0] - trust_region[0], r_x0[0] + trust_region[0]),
                (r_x0[1] - trust_region[1], r_x0[1] + trust_region[1]),
                (r_x0[2] - trust_region[2], r_x0[2] + trust_region[2]),
            ],
            args=args,
            **method_kwargs,
        )
```

```
    elif method.__name__ == "dual_annealing":
        soln = method(
            _orientation_objective_function_euler,
            bounds=[
                (r_x0[0] - trust_region[0], r_x0[0] + trust_region[0]),
                (r_x0[1] - trust_region[1], r_x0[1] + trust_region[1]),
                (r_x0[2] - trust_region[2], r_x0[2] + trust_region[2]),
            ],
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "basinhopping":
        method_kwargs["minimizer_kwargs"]["args"] = args
        soln = method(
            _orientation_objective_function_euler,
            x0=r_x0,
            **method_kwargs,
        )

    score = 1 - soln.fun
    refined_phi1 = soln.x[0]
    refined_Phi = soln.x[1]
    refined_phi2 = soln.x[2]

    return (score, refined_phi1, refined_Phi, refined_phi2)
```

A.2.3 Projection Center Refinement

Code listing A.7: Solver function for projection center refinement.

```
def _refine_pc_solver(
    exp, r, pc, method, method_kwargs, pre_args, trust_region
):
    args = (exp,) + pre_args + (r,)
    pc_x0 = pc

    if method.__name__ == "minimize":
        soln = method(
            _projection_center_objective_function,
            x0=pc_x0,
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "differential_evolution":
        soln = method(
            _projection_center_objective_function,
            bounds=[
                (pc_x0[0] - trust_region[0], pc_x0[0] + trust_region[0]),
                (pc_x0[1] - trust_region[1], pc_x0[1] + trust_region[1]),
                (pc_x0[2] - trust_region[2], pc_x0[2] + trust_region[2]),
            ],
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "dual_annealing":
        soln = method(
            _projection_center_objective_function,
            bounds=[
```

```
                (pc_x0[0] - trust_region[0], pc_x0[0] + trust_region[0]),
                (pc_x0[1] - trust_region[1], pc_x0[1] + trust_region[1]),
                (pc_x0[2] - trust_region[2], pc_x0[2] + trust_region[2]),
            ],
            args=args,
            **method_kwargs,
        )
    elif method.__name__ == "basinhopping":
        method_kwargs["minimizer_kwargs"]["args"] = args
        soln = method(
            _projection_center_objective_function,
            x0=pc_x0,
            **method_kwargs,
        )

    score = 1 - soln.fun
    pcx = soln.x[0]
    pcy = soln.x[1]
    pcz = soln.x[2]
    return (score, pcx, pcy, pcz)
```

A.3 Objective Functions

A.3.1 Full Refinement

Code listing A.8: Objective function for the full refinement.

```
def _full_objective_function_euler(x, *args):
    experimental = args[0]
    master_north = args[1]
    master_south = args[2]
    npx = args[3]
    npy = args[4]
    scale = args[5]
    detector_data = args[6]
    mask = args[7]

    detector_ncols = detector_data[0]
    detector_nrows = detector_data[1]
    detector_px_size = detector_data[2]

    # From Orix.rotation.from_euler()
    phi1 = x[0]
    Phi = x[1]
    phi2 = x[2]

    alpha = phi1
    beta = Phi
    gamma = phi2

    sigma = 0.5 * np.add(alpha, gamma)
    delta = 0.5 * np.subtract(alpha, gamma)
    c = np.cos(beta / 2)
    s = np.sin(beta / 2)

    # Using P = 1 from A.6
    q = np.zeros((4,))
```

```
    q[..., 0] = c * np.cos(sigma)
    q[..., 1] = -s * np.cos(delta)
    q[..., 2] = -s * np.sin(delta)
    q[..., 3] = -c * np.sin(sigma)

    for i in [1, 2, 3, 0]:  # flip the zero element last
        q[..., i] = np.where(q[..., 0] < 0, -q[..., i], q[..., i])

    rotation = q

    x_star = x[3]
    y_star = x[4]
    z_star = x[5]

    xpc = detector_ncols * (x_star - 0.5)  # Might be sign issue here?
    xpc = -xpc
    ypc = detector_nrows * (0.5 - y_star)
    L = detector_nrows * detector_px_size * z_star

    alpha2 = detector_data[3]  # Different alpha
    dc = _fast_get_dc(
        xpc, ypc, L, detector_ncols, detector_nrows, detector_px_size, alpha2
    )

    sim_pattern = _fast_simulate_single_pattern(
        rotation,
        dc,
        master_north,
        master_south,
        npx,
        npy,
        scale,
    )
    sim_pattern = sim_pattern * mask

    result = py_ncc(experimental, sim_pattern)
    return 1 - result
```

A.3.2 Orientation Refinement

Code listing A.9: Objective function for the orientation refinement.

```
def _orientation_objective_function_euler(x, *args):
    experimental = args[0]
    master_north = args[1]
    master_south = args[2]
    npx = args[3]
    npy = args[4]
    scale = args[5]
    mask = args[6]
    dc = args[7]

    # From Orix.rotation.from_euler()
    alpha = x[0]  # psi1
    beta = x[1]  # Psi
    gamma = x[2]  # psi3

    sigma = 0.5 * np.add(alpha, gamma)
```

```
    delta = 0.5 * np.subtract(alpha, gamma)
    c = np.cos(beta / 2)
    s = np.sin(beta / 2)

    # Using P = 1 from A.6
    q = np.zeros((4,))
    q[..., 0] = c * np.cos(sigma)
    q[..., 1] = -s * np.cos(delta)
    q[..., 2] = -s * np.sin(delta)
    q[..., 3] = -c * np.sin(sigma)

    for i in [1, 2, 3, 0]:  # flip the zero element last
        q[..., i] = np.where(q[..., 0] < 0, -q[..., i], q[..., i])

    r = q

    sim_pattern = _fast_simulate_single_pattern(
        r,
        dc,
        master_north,
        master_south,
        npx,
        npy,
        scale,
    )

    sim_pattern = sim_pattern * mask

    result = py_ncc(experimental, sim_pattern)
    return 1 - result
```

A.3.3 Projection Center Refinement

Code listing A.10: Projection center refinement objective function.

```
def _projection_center_objective_function(x, *args):
    x_star = x[0]
    y_star = x[1]
    z_star = x[2]

    experimental = args[0]
    master_north = args[1]
    master_south = args[2]
    npx = args[3]
    npy = args[4]
    scale = args[5]
    detector_data = args[6]
    mask = args[7]
    rotation = args[8]

    detector_ncols = detector_data[0]
    detector_nrows = detector_data[1]
    detector_px_size = detector_data[2]
    alpha = detector_data[3]

    xpc = detector_ncols * (x_star - 0.5)  # Might be sign issue here?
    xpc = -xpc
    ypc = detector_nrows * (0.5 - y_star)
```

```
    L = detector_nrows * detector_px_size * z_star

    dc = _fast_get_dc(
        xpc, ypc, L, detector_ncols, detector_nrows, detector_px_size, alpha
    )

    sim_pattern = _fast_simulate_single_pattern(
        rotation,
        dc,
        master_north,
        master_south,
        npx,
        npy,
        scale,
    )

    sim_pattern = sim_pattern * mask

    result = py_ncc(experimental, sim_pattern)
    return 1 - result
```

A.4 Single Pattern Simulation Functions

A.4.1 Compute Single Pattern

Code listing A.11: Method to simulate a single EBSD pattern.

```
@numba.njit(nogil=True)
def _fast_simulate_single_pattern(
    r,
    dc,
    master_north,
    master_south,
    npx,
    npy,
    scale,
):

    # From orix.quaternion.Quaternion.__mul__

    a = r[0]
    b = r[1]
    c = r[2]
    d = r[3]

    x = dc[..., 0]
    y = dc[..., 1]
    z = dc[..., 2]

    x_new = (a ** 2 + b ** 2 - c ** 2 - d ** 2) * x + 2 * (
        (a * c + b * d) * z + (b * c - a * d) * y
    )
    y_new = (a ** 2 - b ** 2 + c ** 2 - d ** 2) * y + 2 * (
        (a * d + b * c) * x + (c * d - a * b) * z
    )
    z_new = (a ** 2 - b ** 2 - c ** 2 + d ** 2) * z + 2 * (
        (a * b + c * d) * y + (b * d - a * c) * x
```

```
    )
    rotated_dc = np.stack((x_new, y_new, z_new), axis=-1)

    (
        nii,
        nij,
        niip,
        nijp,
        di,
        dj,
        dim,
        djm,
    ) = _fast_get_lambert_interpolation_parameters(
        rotated_direction_cosines=rotated_dc,
        npx=npx,
        npy=npy,
        scale=scale,
    )
    pattern = np.zeros(shape=rotated_dc.shape[0:-1], dtype=np.float32)
    for i in range(rotated_dc.shape[0]):
        for j in range(rotated_dc.shape[1]):
            _nii = nii[i][j]
            _nij = nij[i][j]
            _niip = niip[i][j]
            _nijp = nijp[i][j]
            _di = di[i][j]
            _dj = dj[i][j]
            _dim = dim[i][j]
            _djm = djm[i][j]
            if rotated_dc[..., 2][i][j] >= 0:
                pattern[i][j] = (
                    master_north[_nii, _nij] * _dim * _djm
                    + master_north[_niip, _nij] * _di * _djm
                    + master_north[_nii, _nijp] * _dim * _dj
                    + master_north[_niip, _nijp] * _di * _dj
                )
            else:
                pattern[i][j] = (
                    master_south[_nii, _nij] * _dim * _djm
                    + master_south[_niip, _nij] * _di * _djm
                    + master_south[_nii, _nijp] * _dim * _dj
                    + master_south[_niip, _nijp] * _di * _dj
                )
    return pattern
```

A.4.2 Master Pattern Data Extraction

Code listing A.12: Method to extract data from master patterns.

```
def _get_single_pattern_params(mp, detector, energy):
    # This method is already a part of the EBSDMasterPattern.get_patterns so
    # it could probably replace it?
    if mp.projection != "lambert":
        raise NotImplementedError(
            "Master pattern must be in the square Lambert projection"
        )
    # if len(detector.pc) > 1:
    #     raise NotImplementedError(
```

```
    #         "Detector must have exactly one projection center"
    #     )

    # Get the master pattern arrays created by a desired energy
    north_slice = ()
    if "energy" in [i.name for i in mp.axes_manager.navigation_axes]:
        energies = mp.axes_manager["energy"].axis
        north_slice += ((np.abs(energies - energy)).argmin(),)
    south_slice = north_slice
    if mp.hemisphere == "both":
        north_slice = (0,) + north_slice
        south_slice = (1,) + south_slice
    elif not mp.phase.point_group.contains_inversion:
        raise AttributeError(
            "For crystals of point groups without inversion symmetry, like "
            f"the current {mp.phase.point_group.name}, both hemispheres "
            "must be present in the master pattern signal"
        )
    master_north = mp.data[north_slice]
    master_south = mp.data[south_slice]
    npx, npy = mp.axes_manager.signal_shape
    scale = (npx - 1) / 2

    return master_north, master_south, npx, npy, scale
```

A.4.3 Computation of Direction Cosines with Multiple Projection Centers

Code listing A.13: Method to calculate direction cosines when the experimental geometry has multiple projection centers.

```
@numba.njit(nogil=True)
def _fast_get_dc_multiple_pc(
    xpc, ypc, L, scan_points, ncols, nrows, px_size, alpha
):
    nrows = int(nrows)
    ncols = int(ncols)

    ca = np.cos(alpha)
    sa = np.sin(alpha)

    # 1 DC per scan point
    r_g_array = np.zeros((scan_points, nrows, ncols, 3), dtype=np.float32)
    for k in range(scan_points):
        det_x = (
            -1
            * ((-xpc[k] - (1.0 - ncols) * 0.5) - np.arange(0, ncols))
            * px_size
        )
        det_y = ((ypc[k] - (1.0 - nrows) * 0.5) - np.arange(0, nrows)) * px_size
        L2 = L[k]
        for i in range(nrows):
            for j in range(ncols):
                x = det_y[nrows - i - 1] * ca + sa * L2
                y = det_x[j]
                z = -sa * det_y[nrows - i - 1] + ca * L2
                r_g_array[k][i][j][0] = x
```

```
                r_g_array[k][i][j][1] = y
                r_g_array[k][i][j][2] = z

    norm = np.sqrt(np.sum(np.square(r_g_array), axis=-1))
    norm = np.expand_dims(norm, axis=-1)
    r_g_array = r_g_array / norm

    return r_g_array
```

A.4.4 Computation of Direction Cosines with a single, fixed Projection Center

Code listing A.14: Method to calculate direction cosines when the experimental geometry has a single, fixed projection center.

```
@numba.njit(nogil=True)
def _fast_get_dc(xpc, ypc, L, ncols, nrows, px_size, alpha):
    # alpha: alpha = (np.pi / 2) - sigma + theta_c
    # Detector coordinates in microns
    nrows = int(nrows)
    ncols = int(ncols)
    det_x = -1 * ((-xpc - (1.0 - ncols) * 0.5) - np.arange(0, ncols)) * px_size
    det_y = ((ypc - (1.0 - nrows) * 0.5) - np.arange(0, nrows)) * px_size

    # Auxilliary angle to rotate between reference frames

    ca = np.cos(alpha)
    sa = np.sin(alpha)

    r_g_array = np.zeros((nrows, ncols, 3), dtype=np.float32)

    for i in range(nrows):
        for j in range(ncols):
            x = det_y[nrows - i - 1] * ca + sa * L
            y = det_x[j]
            z = -sa * det_y[nrows - i - 1] + ca * L
            r_g_array[i][j][0] = x
            r_g_array[i][j][1] = y
            r_g_array[i][j][2] = z

    norm = np.sqrt(np.sum(np.square(r_g_array), axis=-1))
    norm = np.expand_dims(norm, axis=-1)
    r_g_array = r_g_array / norm

    return r_g_array
```

A.4.5 Lambert Projection of Direction Cosines

Code listing A.15: Project a vector to the Lambert projection.

```
@numba.njit(nogil=True)
def _fast_lambert_projection(v):
    w = np.atleast_2d(v)
    norm = np.sqrt(np.sum(np.square(w), axis=-1))
    norm = np.expand_dims(norm, axis=-1)
```

```
    w = w / norm

    x = w[..., 0]
    y = w[..., 1]
    z = w[..., 2]

    # Arrays used in both setting X and Y
    sqrt_z = np.sqrt(2 * (1 - np.abs(z)))
    sign_x = np.sign(x)
    sign_y = np.sign(y)
    abs_yx = np.abs(y) <= np.abs(x)

    # Reusable constants
    sqrt_pi = np.sqrt(np.pi)
    sqrt_pi_half = sqrt_pi / 2
    two_over_sqrt_pi = 2 / sqrt_pi

    # Ensure (0, 0) is returned where |z| = 1
    lambert = np.zeros(x.shape + (2,), dtype=np.float32)
    # z_not_one = np.abs(z) != 1

    # I believe it currently returns invalid results for the vector [0, 0, 1]
    # as discussed in https://github.com/pyxem/kikuchipy/issues/272

    # Numba does not support the fix implemented in the main code
    # one workaround could be to implement a standard loop setting the values

    # Equations (10a) and (10b) from Callahan and De Graef (2013)
    lambert[..., 0] = np.where(
        abs_yx,
        sign_x * sqrt_z * sqrt_pi_half,
        sign_y * sqrt_z * (two_over_sqrt_pi * np.arctan(x / y)),
    )
    lambert[..., 1] = np.where(
        abs_yx,
        sign_x * sqrt_z * (two_over_sqrt_pi * np.arctan(y / x)),
        sign_y * sqrt_z * sqrt_pi_half,
    )
    return lambert
```

A.4.6 Lambert Projection Interpolation Parameters

Code listing A.16: Fast method to get the Lambert interpolation parameters.

```
@numba.njit(nogil=True)
def _fast_get_lambert_interpolation_parameters(
    rotated_direction_cosines: np.ndarray,
    npx: int,
    npy: int,
    scale: Union[int, float],
) -> tuple:

    xy = (
        scale
        * _fast_lambert_projection(rotated_direction_cosines)
        / (np.sqrt(np.pi / 2))
    )
```

```
    i = xy[..., 1]
    j = xy[..., 0]
    nii = (i + scale).astype(np.int32)
    nij = (j + scale).astype(np.int32)
    niip = nii + 1
    nijp = nij + 1
    niip = np.where(niip < npx, niip, nii).astype(np.int32)
    nijp = np.where(nijp < npy, nijp, nij).astype(np.int32)
    nii = np.where(nii < 0, niip, nii).astype(np.int32)
    nij = np.where(nij < 0, nijp, nij).astype(np.int32)
    di = i - nii + scale
    dj = j - nij + scale
    dim = 1.0 - di
    djm = 1.0 - dj

    return nii, nij, niip, nijp, di, dj, dim, djm
```

A.5 Similarity Metrics

A.5.1 Normalized Cross Correlation Coefficient

Code listing A.17: Quick NCC calculation.

```
@numba.njit(fastmath=True)
def py_ncc(a, b):
    # Input should already be np.float32
    abar = np.mean(a)
    bbar = np.mean(b)
    astar = a - abar
    bstar = b - bbar
    return np.sum(astar * bstar) / np.sqrt(
        np.sum(np.square(astar)) * np.sum(np.square(bstar))
    )
```

A.5.2 Normalized Dot Product

Code listing A.18: Quick NDP calculation.

```
@numba.njit(fastmath=True)
def py_ndp(a, b):
    # Input should already be np.float32
    return np.sum(a * b) / np.sqrt(np.sum(np.square(a)) * np.sum(np.square(b)))
```

Appendix B

Code Examples

B.1 Scalability Evaluation Script

Code listing B.1: Script to time the performance of the orientation refinement.

```
import numpy as np
from orix import sampling, plot, io
import kikuchipy as kp
from orix.quaternion import Rotation
import time
import os
from kikuchipy.indexing.refinement import Refinement

r = Rotation.random(10000)


mp = kp.data.nickel_ebsd_master_pattern_small(projection="lambert", energy=20)

detector = kp.detectors.EBSDDetector(
    shape=(100, 100),
    pc=[0.421, 0.7794, 0.5049],
    sample_tilt=70,
    convention="tsl",
)

s = mp.get_patterns(
    rotations=r,
    detector=detector,
    energy=20,
    dtype_out=np.uint8,
    compute=True
    )


ni = mp.phase
r_fz = sampling.get_sample_fundamental(
    resolution=4, space_group=ni.space_group.number
)
r_fz

sim = mp.get_patterns(
    rotations=r_fz,
```

```
    detector=detector,
    energy=20,
    dtype_out=np.uint8,
    compute=True
    )

xmap = s.match_patterns(sim, n_slices=10, keep_n=2)

start = time.time()
r_xmap = Refinement.refine_orientations(
            xmap=xmap,
            mp=mp,
            exp=s,
            det=detector,
            energy=20
        )
end = time.time()

print("Cores: ", len(os.sched_getaffinity(0)), "Time: ", end - start)
```

B.2 Effect of Inaccurate Projection Center

Code listing B.2: Simulation of ground truth.

```
import kikuchipy as kp
from orix import sampling
from orix.quaternion import Rotation


detector = kp.detectors.EBSDDetector(
        shape=(60, 60),
        pc=(0.5070, 0.7230, 0.5613),
        px_size=59.2,
        sample_tilt=70,
        convention="tsl"
    )


mp_ni = kp.load("Ni-master.h5",
        projection="lambert",
        hemisphere="both",
        energy=20
    )


r = Rotation.random(1000)


ground_truth = mp_ni.get_patterns(
               rotations=r,
               detector=detector,
               energy=20,
               dtype_out=np.float32,
               compute=True
    )
```

Code listing B.3: DI with erorr in projection center parameters, followed by refinement.

```
r_ni = sampling.sample_generators.get_sample_fundamental(
        resolution=1.4,
        space_group=225
    )

error = 7 # Some number between -7 and 7

detector = kp.detectors.EBSDDetector(
           shape=(60, 60),
           pc=(
           0.5070*(1 + error/100),
           0.7230*(1 + error/100),
           0.5613*(1 + error/100)
           ),
           px_size=59.2,
           sample_tilt=70,
           convention="tsl"
    )


ni_dict = mp_ni.get_patterns(
          rotations=r_ni,
          detector=detector,
          energy=20,
          dtype_out=np.uint8,
          compute=False
    )


di_xmap = ground_truth.match_patterns(
          [ni_dict],
          keep_n=1,
          n_slices=30,
          return_merged_crystal_map=False
    )


o_xmap = Refinement.refine_orientations(
         di_xmap,
         mp_ni,
         ground_truth,
         detector,
         20,
         method_kwargs={'method': "Nelder-Mead"}
    )


opc_xmap, _ = Refinement.refine_xmap(
              di_xmap,
              mp_ni,
              ground_truth,
              detector,
              20,
              method_kwargs={'method': "Nelder-Mead"}
    )
```

B.3 Projection Center Correction Step

Code listing B.4: Projection Center correction factor method.

```
import numpy as np
import kikuchipy as kp

def pc_correction(detector, sx, sy, stepsize):
    rows, cols = detector.shape
    pc_estimate = detector.pc_emsoft()
    xpc = -pc_estimate[...,0][0]
    ypc = pc_estimate[...,1][0]
    L = pc_estimate[...,2][0]

    delta = stepsize/detector.px_size

    theta_c = np.radians(detector.tilt)
    sigma = np.radians(detector.sample_tilt)
    alpha = (np.pi / 2) - sigma + theta_c

    ca = np.cos(alpha)
    sa = np.sin(alpha)

    new_xpc = ((1-sx) * 0.5 + np.arange(0, sx))
    new_xpc = xpc - (new_xpc*delta)
    new_xpc = np.tile(new_xpc, (sy, 1))
    new_xpc = np.ravel(new_xpc)

    new_ypc = -((1-sy) * 0.5 + np.arange(0, sy))
    new_ypc = ypc - (new_ypc * ca*delta)
    new_ypc = np.transpose([new_ypc] * sx)
    new_ypc = np.ravel(new_ypc)

    new_L = -((1-sy) * 0.5 + np.arange(0, sy))
    new_L = L - (new_L * sa * delta * detector.px_size)
    new_L = np.transpose([new_L] * sx)
    new_L = np.ravel(new_L)
    new_pc = np.column_stack((new_xpc, new_ypc, new_L))

    new_detector = kp.detectors.EBSDDetector(
        shape=(rows, cols),
        pc=new_pc,
        px_size=detector.px_size,
        sample_tilt=detector.sample_tilt,
        tilt=detector.tilt,
        binning=detector.binning,
        convention="emsoft4"
    )

    return new_detector
```

B.4 Orientation Correction Step

Code listing B.5: Orientation correction factor method.

```
from orix.quaternion import Rotation

def orientation_correction_factor(rxmap, rows, cols, org_det):
    old_rdata = rxmap.rotations.data.reshape((rows, cols, 4))
    new_rdata = np.zeros_like(old_rdata)

    old_rotations = Rotation(old_rdata)

    M = rows + 1
    N = cols + 1

    M = M // 2
    N = N // 2

    L = org_det.pc_emsoft()[..., 2][0]
    delta = org_det.px_size

    sigma = np.deg2rad(org_det.sample_tilt)
    theta_c = np.deg2rad(org_det.tilt)

    alpha = np.pi / 2 - sigma + theta_c

    ca = np.cos(alpha)
    sa = np.sin(alpha)
    # https://github.com/EMsoft-org/EMsoft/blob/develop/Source/
    # EMsoftHDFLib/EBSDDImod.f90#L810
    # (J. Appl. Cryst. (2017). 50, 1664  1676  , eq.15)

    initialx = 49
    initialy = 74

    i = np.arange(1, rows+1)
    j = np.arange(1, cols + 1)

    stepx = 20
    stepy = 20

    dpcx = - ( initialx - j) * stepx
    dpcy = - ( initialy - i) * stepy

    # Shift to the detector reference frame and px units
    dpcx = - dpcx / delta
    dpcl = - dpcy * sa
    dpcy = - dpcy * ca / delta

    for i in range(rows):
        for j in range(cols):
            dx = dpcx[j]
            dy = dpcy[i]
            if (dx != 0) or (dy != 0):
                current = old_rotations[i, j]
                rho = dx**2 + dy**2
                nn = -1* np.array((dx*ca, -dy, -dx*sa)) / np.sqrt(rho)
                omega = np.arccos(L / np.sqrt(L**2 + delta**2 * rho))
                nnn = np.sin(omega*0.5) * nn
                qq = np.array((np.cos(omega*0.5), nnn[0], nnn[1], nnn[2]))
                temp = Rotation(qq)
                new_r = current * temp
                new_rdata[i, j] = new_r.data
            else:
```

```
                new_rdata[i, j] = old_rdata[i, j]

    corrected_rotations = Rotation(new_rdata)
    return corrected_rotations
```

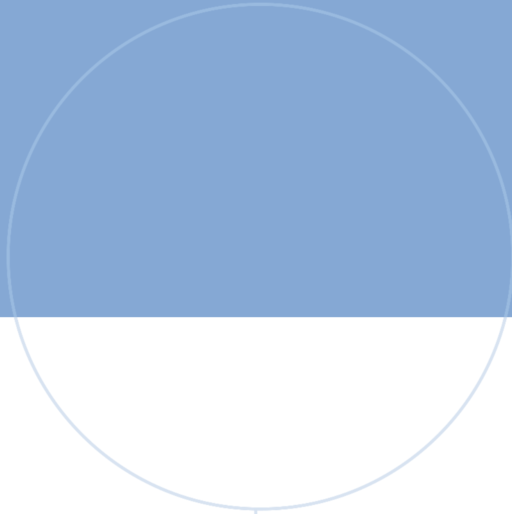
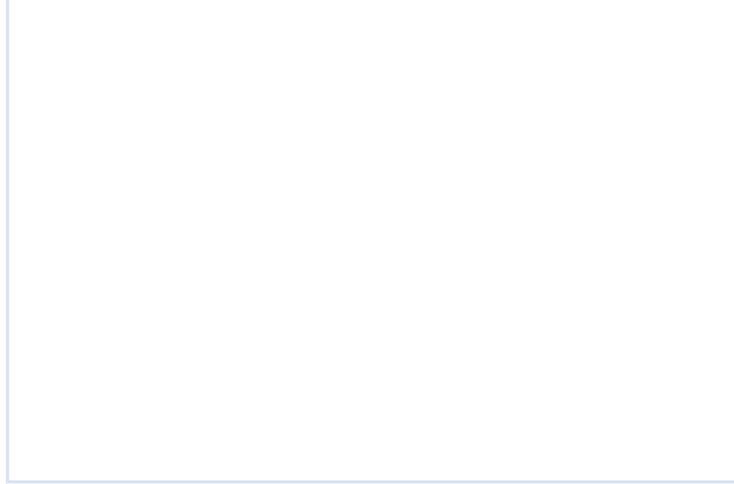
B.5 scikit-image Mask from Experimental Pattern.

Code listing B.6: Creation of threshold mask using scikit-image.

```
from skimage.filters import threshold_triangle

# Raw Experimental Data
image = experimental.data[24, 24]

thresh_tri = threshold_triangle(image)
si_mask = image > thresh_tri
```

NTNU

Norwegian University of
Science and Technology