LA-UR-16-25109 (Accepted Manuscript)

Monte Carlo modeling of recrystallization processes in -uranium

Steiner, Matt A
Mccabe, Rodney James
Garlea, Elena
Agnew, Sean R

To be published in: Journal of Nuclear Materials

DOI to publisher's version: 10.1016/j.jnucmat.2017.04.026

Permalink to record: http://permalink.lanl.gov/object/view?what=info:lanl-repo/lareport/LA-UR-16-25109

Accepted Manuscript

Monte Carlo modeling of recrystallization processes in α-uranium

M.A. Steiner, R.J. McCabe, E. Garlea, S.R. Agnew

PII: S0022-3115(16)31021-2

DOI: 10.1016/j.jnucmat.2017.04.026

Reference: NUMA 50246

To appear in: *Journal of Nuclear Materials*

Received Date: 26 October 2016

Revised Date: 23 March 2017

Accepted Date: 20 April 2017

Please cite this article as: M.A. Steiner, R.J. McCabe, E. Garlea, S.R. Agnew, Monte Carlo modeling of recrystallization processes in α-uranium, *Journal of Nuclear Materials* (2017), doi: 10.1016/j.jnucmat.2017.04.026.

Monte Carlo Modeling of Recrystallization Processes in α-Uranium

M.A. Steiner*[1], R.J. McCabe[2], E. Garlea[3], and S.R. Agnew[1]

[1] *University of Virginia, Material Science and Engineering, 395 McCormick Rd, Charlottesville, VA 22904, USA*
[2] *Materials Science and Technology Division, Los Alamos National Laboratory, Los Alamos, NM, USA*
[3] *Y-12 National Security Complex, Oak Ridge, TN 37831, USA*

Abstract

Starting with electron backscattered diffraction (EBSD) data obtained from a warm clock-rolled α-uranium deformation microstructure, a Potts Monte Carlo model was used to simulate static site-saturated recrystallization while testing a number of different conditions for the assignment of recrystallized nuclei within the microstructure. The simulations support observations that recrystallized nuclei within α-uranium form preferentially on non-twin high-angle grain boundary sites at 450°C, and demonstrate that the most likely nucleation sites on these boundaries can be identified by the surrounding degree of Kernel Average Misorientation (KAM), which may be considered as a proxy for the local geometrically necessary dislocation (GND) density.

Keywords: α-uranium, recrystallization, Monte Carlo, Potts modeling, EBSD

1. Introduction

Uranium components are typically produced in a similar manner to many other metals, starting from cast material that undergoes a series of thermomechanical processes such as forging, rolling, swaging and forming. Deformation processing steps are frequently interspersed with heat treatments to produce finer, recrystallized grains and restore ductility. Along with having significant effects on grain size and morphology, thermomechanical processes can cause significant evolution in a material's crystallographic texture. The crystallographic texture of uranium is often of critical interest because the low temperature allotrope, the orthorhombic α-

phase, has unique anisotropic properties including a negative thermal expansion coefficient along one crystallographic direction [1 - 4]. A number of investigations in the last decade have explored the microstructural and textural evolution of α-uranium during different types of deformation [5 - 12]. Post-deformation recrystallization phenomena and associated modeling approaches are well documented for a number of predominantly cubic metals [13 - 17], but comparatively less study has been made into the recrystallization processes governing metals of lower symmetry. Lower symmetry crystal structures generally deform via several distinct modes of dislocation slip and deformation twinning, the relative activities of which are sensitive to grain orientation and temperature. These slip and twinning modes certainly control the deformation texture evolution, and in turn could impact the texture evolution during recovery and recrystallization. This is especially true for α-uranium, which has four distinct dislocation slip modes and at least three modes of deformation twinning, many with multiple slip systems or variants [6, 18, 19]. While it is likely that recrystallization in α-uranium proceeds through a process comparable to that of higher symmetry metals, efforts to establish the exact nature of this process have been hampered by the limited amount of experimental data available in the literature. A number of empirical studies on the recrystallization of α-uranium were carried out into the early 1960s, but the crystallographic texture analysis was rudimentary [20 - 27] and the data generated in these studies cannot be correlated to individual grains in a manner necessary for developing mechanistic models or improving upon the existing phenomenological understanding. It was established, however, that the crystallographic texture of annealed α-uranium is dependent on the recrystallization temperature [20].

Static recrystallization occurs by the migration of high-angle grain boundaries (HAGBs) driven by the release of stored deformation energy, leading to a new microstructure of relatively

defect-free grains. It has been established that the nuclei of recrystallized grains already exist in deformed microstructures (or coalesce prior to recrystallization during static or dynamic recovery) [13 - 17]. The nuclei do not always originate from the dominant crystallographic orientations of the deformed microstructure, and thus, their growth can lead to texture evolution. Both the formation of these recrystallized nuclei and the stored energy that induces their expansion through the deformed microstructure are heavily impacted by the type and degree of deformation, as well as grain orientation, size, shape, and distribution. To establish an understanding of recrystallization in α-uranium on a level comparable to cubic metals, detailed microstructural analysis of deformed and subsequently recrystallized samples is necessary.

Electron backscattered diffraction (EBSD) is an ideal tool for characterizing recrystallization as it provides statistically significant, spatially sensitive crystallographic information at the sub-granular scale. This allows for construction not only of the sample's collective crystallographic texture, but also structural features at specific locations, including individual grain orientations, grain boundary geometry, and intra-granular misorientation due to deformation. Successful preparation of α-uranium samples for EBSD analysis is notoriously difficult [28 - 30]. In addition to radiological safety concerns, the surface of uranium metal rapidly oxidizes upon exposure to air. The challenges of oxidation are exacerbated by the high atomic number of uranium, which restricts the interaction volume of backscattered electrons and prevents interrogation of the metal beneath all but the thinnest of oxide layers. Despite these difficulties, high quality EBSD data of warm-rolled α-uranium before, during, and after recrystallization at 450°C has recently been published by one of the present authors [31]. Comparing the texture of the recrystallized microstructure to that of select components of the deformed microstructure, it was noted that the varying degrees of deformation present in grains

of different orientations does not immediately explain the observed crystallographic texture evolution during recrystallization [31]. Instead, the collective texture of points found along non-twin HAGBs is a closer match to the recrystallized texture, suggesting that recrystallized nuclei at 450°C form primarily along these boundaries.

It is possible to simulate recrystallization within a microstructure characterized by EBSD through a number of different techniques, including cellular automata, phase field, and Monte Carlo models [32 - 34]. Often, one of the biggest challenges for all of these models is determining the nucleation parameters governing the formation and location of recrystallized nuclei [34], which are hard to determine *a priori* and must be informed by comparing experimental and simulated results. The simulations in the present study were performed utilizing a Monte Carlo (MC) technique known as Potts modeling. MC Potts modeling is a generalization of the simpler Ising model to systems with more than two states (e.g. unique grain orientations) [35 - 37], and has long been used to simulate grain growth phenomena within recrystallized microstructures [35, 36]. Subsequent expansions of the model have been successfully employed to simulate recrystallization as well as grain growth, starting from deformed microstructures [38 - 41].

The Potts model begins with a 2D or 3D lattice, where each point is assigned to a specific grain and the lattice is sized so that each grain contains a significant number of lattice points at initialization. Driven by a minimization of energy, the Potts model uses probabilistic methods to test small changes on the local scale, preferentially keeping changes that decrease the system's energy. When modeling recrystallization through the Potts framework, the total energy of the system can be expressed by the following Hamiltonian:

Eq. 1 $$H = \sum_{i=1}^{N} \left\{ E_i + \frac{1}{2} \sum_{j=1}^{Z} \gamma_{ij} \right\}$$

where E_i is the stored energy at site i (above a recrystallized ground state), γ_{ij} is the grain boundary energy between sites i and j summed over Z nearest neighbors (halved to prevent double counting), and both energy contributions are summed over all N-sites [40].

A simulation microstructure in the Potts model can be populated directly from experimental EBSD data, allowing each lattice site to be assigned a unique set of Euler angles according to the measured orientation. For a 2D square lattice, each lattice point can be considered to share 8 neighbors (including diagonals). When two neighboring lattice sites in the simulation are assigned to different grains, their interface represents a grain boundary segment with an associated grain boundary energy. For most materials, the boundary energy can be defined as a function of the grain misorientation, based on the Read-Shockley model [42, 43] together with the energies of special boundaries corresponding to the coincident site lattice model and additional elements as necessary.

If the simulation is being initialized by a computational technique (e.g. an N-site VPSC model [39], finite element [41], or fast Fourier transform based [44] full-field crystal plasticity model) the stored energy from deformation, E_i, is straightforward. The assignment of stored energy to lattice sites based on experimental data is more complicated and requires extraction of quantities related to deformation from the EBSD map. The Image Quality (IQ) factor is a value that measures the sharpness of bands in the backscattered electron Kikuchi pattern used to index EBSD patterns. IQ is affected by gradients in the amount of local elastic strain present at each lattice point, allowing it to detect the difference between recrystallized and non-recrystallized grains [45 - 48], and it is possible to derive an empirical relationship between IQ and the stored elastic energy for a material [40]. Stored energy can also be related to the Kernel Average Misorientation (KAM) of each lattice point in the structure [49 - 57]. KAM is defined for a

given lattice point as the average misorientation from other points of the same grain located inside a surrounding kernel of specified size, e.g. a second nearest-neighbor 5x5 grid, and correlates with the underlying density of dislocations. When utilizing either IQ or KAM, the measured values must be placed in units of energy that are properly scaled to the grain boundary energy γ, and both techniques have inherent problems; IQ will vary with factors such as the surface quality of the prepared sample, EBSD parameters, and EBSD pattern filtering, while KAM only permits an indirect (though often accurate [56, 57]) approximation of the total stored dislocation density. In the case of α-uranium, which has propensity to rapidly oxidize and requires EBSD pattern filtering, KAM becomes the logical proxy for stored energy as it is less dependent than IQ on the surface state of the material [30].

Once a microstructure and its governing energetics have been established, the Potts model executes, in series, a string of independent "events" at discrete lattice points within the microstructure, following a randomized order with one event executed at each site per simulation time step. During each event, the simulation will randomly select a lattice site neighboring the event site and test the change of energy that would result from replacing the event site with the neighboring site's assignment, preferentially keeping the replacement and growing the neighboring grain when it decreases the energy of the system ($\Delta H < 0$). Through this stochastic process, the model is able to simulate evolution of the experimental microstructure and the resulting crystallographic texture. There are, however, several deficiencies to the standard formulation of the MC Potts model, including curvature artifacts related to the lattice, scaling of simulation units to physical units, and the concepts of real time and temperature not being explicitly tracked or handled. Improved formulations of the MC Potts model have been introduced to handle each of these issues and promote its predictive power [58 - 61].

This paper uses small modifications to the standard implementation of the Potts MC model and EBSD data from warm-rolled α-uranium originally published in Ref. [31] (Fig. 1) to simulate textures that result from different assignments of recrystallized nuclei within the deformed microstructure. By comparing the textures simulated from different nuclei assignments to experimental recrystallization textures, the most likely location of recrystallized nuclei in the warm clock-rolled α-uranium microstructure are determined. The specific conditions used to best identify the nuclei can then be used to inform how recrystallization progresses in α-uranium deformation microstructures simulated using crystal plasticity based computational techniques.

2. Experimental methods

The experimental EBSD maps presented within this paper were acquired during the study in Ref. [31]. The α-uranium samples were warm clock-rolled at 300°C in eight equivalent strain passes, with rotations of 0°, 90°, 135°, 225°, 270°, 360°, 45°, and 135°, resulting in a final reduction of approximately 50%. Clock-rolling is known to produce less in-plane texture anisotropy than unidirectional rolling of uranium [62] [63]. The warm-rolling temperature was considerably below the threshold that static recrystallization is observed to occur within uranium samples rolled to similar levels of deformation; with recrystallization taking approximately 100 hr. to complete at 380°C and becoming kinetically limited below 350°C [20]. In addition, dynamic recrystallization is not expected below 400°C [64]. The recrystallized condition presented was heat treated under vacuum in a quench dilatometer at 450°C for 10^5 seconds. To match the input format of the MC code, the EBSD data was transformed from a hexagonal to a square grid using Oxford Instruments HKL Channel 5 software, with spatial aliasing during conversion acting as a high-pass filter. Analysis of experimental and simulated EBSD data was carried out using the MTeX MATLAB toolbox [65].

3. Computational methods

The simulations conducted for this paper were performed using modifications to a standard implementation of Potts MC code developed for grain growth simulations, outlined in Ref. [66]. A complete version of the modified code has been made available as open source *Supplementary material*. The most significant complication in adopting the standard Potts model to include recrystallization arises during scenarios where the neighboring site selected during an event has not yet recrystallized ($E_i > 0$). The dislocation network responsible for stored energy will not move along with evolving grain boundaries in a deformed microstructure, and a grain boundary moving through a site will clear it of its existing dislocations (setting $E_i = 0$). As a consequence, any instance during which a site is reassigned to the grain of a non-recrystallized neighboring site will create a new recrystallized nucleus via the bulge mechanism, a process typically limited to low-temperature or high-stress conditions [67]. To model recrystallization via site-saturated preexisting nuclei, as expected for α-uranium, the deformed sites in the simulation must be held static [39] and all events where the selected neighboring assignment belongs to a non-recrystallized grain terminated with no change to the microstructure.

When a Potts recrystallization model is initiated from experimental EBSD data, it is necessary to introduce a scaling factor between the boundary energy, γ_i, and the experimentally determined stored energy values, E_i, so that both are in commensurate units. As this scaling factor is difficult to calculate, it can be noted that in sufficiently deformed systems the stored strain energy is much larger than the interfacial energy ($E_i >> \gamma_i$). Therefore, as a first-order approximation, this difference in magnitude conveniently eliminates the need to determine a scaling factor as, in this limit, any event where a recrystallized grain expands into a deformed site has $\Delta H < 0$ and the release of stored energy always overcomes the creation of any new

boundaries. This approximation yields a particularly straightforward adaptation of the standard Potts model to the recrystallization process; where a recrystallized grain will always expand into a deformed site if presented the opportunity, deformed sites are otherwise held static, and standard grain growth behavior can occur between recrystallized grains that have already impinged upon one another.

Due to differences in the driving forces behind recrystallization and standard grain growth, boundaries moving in response to recrystallization will have a considerably higher velocity. As a boundary is only able to travel one lattice site during an event in the simulation, differences in boundary velocity are accounted for statistically over a number of events by scaling the probability of each type of event succeeding. While eliminating grain growth entirely is a reasonable approximation for the simulations in this study, given the relative speed at which recrystallization occurs in α-uranium at 450°C [20], the statistical nature of the MC model produces ragged unphysical grain boundaries in the microstructure if grain growth is completely ignored. A reduced grain growth probability was utilized in this study, where only one fifth of energetically successful attempts (which occur infrequently if the grain boundary is smooth) were allowed to proceed. This was sufficient to provide more realistic boundaries in the microstructure but had no discernable impact on the texture compared to recrystallization simulations run with grain growth completely ignored.

Allowing recrystallized grains to expand into deformed sites with 100% probability effectively fixes the velocity of the recrystallization boundaries to a constant and treats the deformed microstructure as a mean field. Other Potts model based simulations have considered modifications to the recrystallization probability during each event based on the stored energy released by the deformed site (ΔE_i) [38 - 40], as the velocity of a recrystallizing boundary will be

proportional to the driving force [68]. Comparing the number of simulation steps with the experimentally observed time to complete recrystallization [31] reveals that the recrystallization boundary velocity is approximately 5 nm/s, with 1 site = 0.5 μm and 1 time step ≈ 100 s. This is consistent with the largest grains (which have grown with the least impingement) in the experimental recrystallized structure reaching a maximum radius of 50 μm in 10,000 s. For comparison, grain growth in α-uranium occurs at a rate of approximately 0.05 nm/s at 550°C [69], and will be even more negligible at 450°C.

The advantage of reducing the Potts recrystallization model to this simple form is that reasonable predictions of recrystallization textures can be made without focusing intensively on establishing a large number of model parameters. A number of different criteria for assigning recrystallized nuclei within the deformed α-uranium microstructure were tested in an effort to find the best match to the experimental recrystallization texture. Recrystallization "nuclei" in a site-saturated model do not arise from a random thermally activated stochastic process; rather, they are already present within the microstructure as sub-grains, supplying that the EBSD pattern is acquired with a suitable spatial resolution to capture them (after some recovery, as in the present study [31]). Designating nuclei as recrystallized in the model requires assigning an energy of $E_i = 0$ to the desired sites at the start of the simulation, allowing them to inherit the crystallographic orientation of the site in the deformed state [40]. The nucleation criterion that provides the closest match to the experimental recrystallization texture will identify the conditions that make a given sub-grain more likely than others in the microstructure to becoming a recrystallized nucleus.

Recrystallized nuclei often form preferentially at heterogeneities within a microstructure such as grain boundaries. Several nucleation conditions related to HAGBs were tested, such as

including or excluding twin boundary sites. For the purposes of this paper, the definition of a HAGB was chosen as a boundary with misorientation > 20°, as the deformed α-uranium microstructure exhibits considerable intragranular misorientations. Significant misorientation variations also occur along twin boundaries, so "twins" were categorized very simplistically as any boundary within a 60°- 80° misorientation window. A map of HAGBs and twin boundaries, following this designation, is presented in Fig. 2. It can be seen that this conservative twin identification strategy is sufficiently broad to categorize boundary types in the microstructure. A model refinement to handle twin identification would need to be adapted if that were not the case [70]. Grain boundary sites were defined as being separated from a minimum of two neighboring sites and no more than six by a qualifying misorientation angle; suppressing nucleation from experimental noise in the raw EBSD data.

Another set of nucleation conditions was employed to test whether nucleation occurs preferentially at points in the microstructure with large stored energies. For the purposes of this study, stored energy was determined by KAM values using a 3x3 grid with a 10° grain boundary threshold. This choice of KAM kernel assumes the recrystallized nuclei are not larger than a single site (at the 0.5 μm step size), and focuses only on the misorientation from the most immediate neighbors. The 10° threshold likely subsumes some low angle grain boundaries into the KAM, but is necessary due to the degree of intragranular misorientation present in the microstructure. A KAM map of the deformed microstructure is presented in Fig. 3. KAM values are heterogeneously distributed about the microstructure and several regions are comparatively low in deformation. Sites with the highest KAM tend to appear in clusters, strings, or bands located within areas of the deformed microstructure with a large number of non-twin HAGBs. The distribution of KAM values is approximately log-normal (Fig. 4). About 3.5% of sites in the

deformed microstructure are calculated to have a KAM of 0, which occurs due to experimental noise creating points that have greater than 10° misorientation with all neighboring sites within the kernel. As recrystallized grains are defined by $E_i = 0$ for modeling purposes, these grains were assigned $E_i > 0$ to ensure that no unintentionally recrystallized sites were created. A probability function has been proposed for handling nucleation base on stored energy conditions in Potts recrystallization models:

Eq. 2 $$n_i = 1 - \exp\left(-\frac{E_i - E_1}{E_2 - E_1}\right)$$

where E_1 is a threshold of stored energy that E_i must exceed for nucleation to take place, and E_2 is a parameter that controls the form of the function [39]. In the case that E_2 approaches E_1, the nucleation probability reduces to a simple threshold. A simple threshold condition was utilized in this study because the available EBSD data is not large enough to meaningfully fit a set of E_1 and E_2 parameters.

All simulated results presented in this paper were executed over 100 MC time steps unless otherwise noted, which was sufficient for the simulated microstructure to completely recrystallize. Three separate simulations were run at every nucleation condition and averaged to provide a representative crystallographic texture. Periodic boundary conditions were imposed to address lattice sites at the edges of the EBSD data, as the percentage of grains that contact the boundary at initialization is sufficiently small to reduce any associated artifacts. Orientation distribution functions were calculated from the experimental and simulated microstructures with a 5° angular resolution using the MTeX software package.

4. Results

The deformed microstructure of a warm clock-rolled α-uranium sample (Fig. 1) was used to initialize all of the simulations presented within this paper. Numerous thin and often

serpentine deformation twins of several modes are present within most grains. Significant (>10°) intra-granular misorientation gradients occur within both the parent grains and twins. The map is orientated in the transverse direction (TD), and grains appear elongated along the rolling direction (RD). The crystallographic texture of the deformed microstructure is presented as pole figures in units of multiples of a random distribution (MRD). Due to the limited number of grains collected in an EBSD scan, the rolled plate's orthotropic symmetry has been imposed on the pole figures presented throughout this paper. Pole figures are all presented in the form of stereographic equal angle projections to match the format of previous uranium recrystallization studies [20]. In addition, the complete orientation distribution functions (ODFs) used to compute all of the pole figures in this paper can be found as *Supplementary material*.

In order to unambiguously discuss the crystallographic texture, it is advantageous to introduce a graphical legend of distinct texture elements (*) that appear repeatedly between the deformed and recrystallized textures (Fig. 5). The crystallographic texture of warm clock-rolled uranium (Fig. 1) is characterized by a strong alignment of (010) poles along RD (*C, and the more diffuse *D), and collection of (001) poles near the normal direction (ND) with a clear splitting along TD (*G). If the texture were solely concentrated at elements *C in the (010) and *G in the (001) pole figures, geometry would dictate the presence of strong elements at *B in (100). Instead, *B is relatively weak compared to the stronger four-fold element *A in the (100) pole figure in the deformed state. These *A features correspond geometrically to the arm-like *E elements in the (010) pole figure, rotating about axes defined by the two very strong *G texture elements. Other features of note are a weak *F texture in the (010) pole figure, and a smooth, strong intensity across features *H-*K in (001).

For comparison, Fig. 6 presents an EBSD map of the clock-rolled material recrystallized at 450°C for 10^5 seconds and determined to be 91 % transformed (with the remaining percentile being highly recovered) [31]. The deformation twins and intragranular misorientation gradients prevalent in the deformed microstructure have been replaced by a finer grained microstructure of relatively defect-free grains. There is notable clustering of like-sized grains in bands along RD. This clustering is attributed to the inhomogeneous distribution of recrystallization nuclei in the deformed microstructure [31]. The observed texture agrees well with previously published textures for similar recrystallized warm clock-rolled material (50% reduction at 330°C, recrystallization at 550°C [7]). Overall the texture is considerably weaker than that of the deformed state. This is best observed in the (001) pole figure where the split TD texture (*G) drops from a maximum of 10 to 5 MRD, elements *H-*J disappear, and the intensity spreads outward in a diffuse halo across much of the pole figure. The ring-like *K element remains weakly present, strengthening where it connects the prominent *G elements. In the (010) pole figure, the axes remain preferentially oriented along the rolling direction, but the maximum intensity has weakened from 7 to 4 MRD, and the stronger *C element has disappeared into the more diffuse *D, while the arm-like *E extensions have retreated. With the loss of *E in the (010) pole figure, the associated *A elements of the (100) are much lower in intensity, while *B has maintained its original intensity and has become the strongest remaining feature. Both the *A and *B elements in the (100) pole figure, as well as the strong *G element in the (001), have experienced a geometrically correlated angular spread along the TD direction (rotating grain orientations about an axis defined by the (010) *C and *D elements along RD). Element *F has disappeared completely in the recrystallized microstructure.

The goal of this study was to identify sites in the deformed microstructure that are the most likely to form recrystallized nuclei, and thereby discern factors influencing recrystallization in the α-uranium system. To identify these sites, a number of different assignment conditions were tested using the Potts recrystallization model; namely HAGBs sites excluding and including twin boundaries, sites with KAM above different threshold values, or combinations of several of these conditions. The simulated textures from these different nuclei assignments were then compared to the experimental recrystallization texture to determine the closest match and any informative trends. For labeling purposes, the HAGB condition includes only the boundary sites designated as HAGBs in Fig. 2 (excluding twin boundaries), while the HAGB + Twins condition includes all of the boundary sites. For each condition nuclei were placed only at sites that simultaneously satisfied all applicable conditions related to the location of grain boundaries (Fig. 2) and their KAM values (Fig. 3). The number of nuclei assigned per simulation in the presented work was held constant at ~1200, regardless of the nucleation conditions, by applying a probability of success to each nucleation site based upon the total number of qualifying nuclei. This cap is the approximate number of nuclei needed to reproduce the observed experimental recrystallized grain size, and also matches the number of nuclei found in the condition that provided the best texture prediction.

It was hypothesized in the original EBSD study that recrystallized nuclei form along non-twin HAGBs in the deformed microstructure at 450°C [31]. The closest match to the experimental crystallographic texture (over an entire microstructure) occurs when the selected nuclei are non-twin HAGB sites with KAM > 9, suggesting that nucleation does not occur homogeneously along these boundaries. The simulated textures from a selection of the explored nucleation conditions are compared in Fig. 7. Each of these conditions will be discussed in the

following paragraphs. Difference pole figures from the experimental recrystallized texture for all conditions can be found as *Supplementary material*. To aide in interpretation of Fig. 7, Fig. 8 graphically displays the strength of the distinct texture elements identified in Fig. 5 for each of the pole figures. Depending on the texture element, the MRD values in Fig. 8 are either the maximum (*A, *B, *F, *G, *H) or minimum (*J) found within each element, or the value at a specific representative location (*C, *D, *E, *I, *K) marked in Fig. 5. It is important to note the size and importance of each element is not equal, and these numbers do not capture changes in the spread or shape of the elements that are best observed in the pole figures themselves. Nonetheless, this semi-quantitative chart helps to visualize related trends and variations between the pole figures.

The HAGB: KAM > 9 condition for recrystallized nuclei (Fig. 9) is able to replicate many of the differences experimentally observed between the deformed and recrystallized textures. In the (010) pole figure element *F has disappeared, both the *C and *D elements in (010) have weakened from the deformation texture, and the arm-like *E extensions are less prominent than in the deformed state. In the (010) pole figure, the primary difference between the simulated and experimental textures is the concentration of intensity directly along RD (*C) rather than being spread across the broader *D element. As with the experimental texture, the loss of *E in the (010) pole figure has led to a corresponding decrease in the associated *A elements of the (100) figure, leaving *B as the strongest remaining feature. Larger disparities occur in the (001) pole figure, despite the simulated and experimental textures undergoing a qualitatively similar evolution. The strong *G texture of the deformed microstructure remains the most intense element and has decreased in intensity, but not by the same extent as the

experimental observation. Elements *H-*K have lost significant portions of their original intensity, but unlike the experimental texture, they have not completely disappeared.

The simulated microstructure in Fig. 9 produces the clustering of grains of similar size seen in the experimental microstructure (Fig. 6). This clustering arises from a heterogeneous density of recrystallized nuclei within the deformed microstructure. Larger recrystallized grains are located in regions of lower deformation (Fig. 3), as the lower density of resulting nuclei will not impinge upon each other immediately during growth. Conversely, regions with smaller grain sizes correspond to areas of greater initial deformation, where the density of nuclei is higher and there is little room for each nucleus to grow. Histograms of the experimentally measured and simulated grain sizes both exhibit log-normal distributions, but the mean grain size of the simulated microstructure is larger (164 μm^2) than that of the experimental (100 μm^2). Differences in the grain size distributions are consistent with effects introduced by limiting the simulation to 2D. In a 3D microstructure grains nucleated above and below the 2D plane will result in a smaller cross-sectional area per grain for a given density of nuclei. While modeling 3D microstructures would offer improvements, the intensive characterization needed to acquire such data is not always feasible. 2D simulations remain an efficient means for studying phenomena and establishing support for experimentally verified trends.

Relaxing the HAGB: KAM > 9 condition to include twin boundary sites or all points in the microstructure with KAM > 9 produces two nearly-identical simulated textures, both of which are similar to the HAGB: KAM > 9 condition. This is because nearly all sites with KAM > 9 occur along boundaries within the microstructure, so removing the requirement to be located on a grain boundary makes little difference. The two alternative KAM > 9 conditions both have textures with weaker *B elements relative to HAGB: KAM > 9, stronger *D elements, and *J

and *K elements that have not reduced to the same extent. All of the KAM > 9 conditions, however, are able to replicate the most notable evolutions from the deformed microstructure: weakening of all elements in the (010) and (001) pole figures, including the near disappearance of *F and *J, and the retreat of elements *E and *A in favor of *B. The importance of including a KAM nucleation condition is highlighted by the textures simulated using only boundary conditions, HAGB or HAGB + Twins. In both of these cases all the elements of the (001) poles are nearly unchanged from the deformed state. The *E element of the (010) pole figure remains pronounced, and the *F element that disappears experimentally remains close to its original strength. The (100) *B element becomes weaker than even the deformed state. Thus, nucleation using only HAGB conditions is unable to fully account for all aspects of the observed texture evolution in α-uranium during recrystallization. The inclusion or exclusion of twin boundaries as potential nucleation sites results in relatively subtle variations in the simulated textures.

Maintaining the non-twin HAGB condition and adjusting the KAM threshold to different values, it can be seen that the KAM > 9 threshold provides the closest texture to the experimental result. With increasing KAM thresholds, the prominent *G, *J and *K elements of the (001) pole figure steadily reduce in spread and intensity, while the relatively minor *I and *H elements exhibit no clear trend. All threshold conditions below KAM > 9 exhibit stronger *E and *D elements in (010) than the KAM > 9 condition. With decreasing KAM thresholds, the *A element of the (100) grows increasingly prominent and exhibits less spread. Finally, applying pure grain growth modeling in the absence of recrystallization to the microstructure and running the simulation until reaching a comparable grain size (600 MC time steps) results in a texture that is similar to the HAGB + Twins condition, with the exception of a stronger *I and weaker *J

element in the (001) pole figure. All of these trends support the conclusion that grain boundary sites with the highest KAM values serve as recrystallization nuclei.

5. Discussion

Textures simulated by nucleation along non-twin HAGBs in the deformed α-uranium microstructure approach the experimental recrystallization texture as previously hypothesized [31], but only by identification of preferential sites with large stored energies (KAM > 9). KAM thresholding actually appears to be the more important to identifying the nucleation sites than the presence of a HAGB, as the textures generated using only a KAM > 9 threshold are closer to the experimental texture than those generated using only the HAGB condition. This is partly due to the fact that KAM > 9 sites exist almost exclusively as a subset of HAGB sites so the conditions cannot be separated. While the simulated results are able to capture a large portion of the α-uranium recrystallization behavior, failure of the simulated recrystallization texture (Fig. 9) to completely replicate the experimental texture (Fig. 6) suggests that improvements can still be made to the model.

One major difference remaining between the simulated and experimental textures is the continued presence of the *H, *I and *K elements of the (001) pole figure. By separating the simulated HAGB: KAM > 9 texture into contributions provided by recrystallized grains of difference sizes, it is clear that the *H, *I and *K elements originate from only the largest ~15% of grains in the microstructure (Fig. 10, Grains > 300 μm^2), which make up about half the areal fraction. Large grains in the simulation form as clusters within the least deformed regions, where a low density of recrystallized nuclei allows for the few existing nuclei to expand significantly before impinging upon each other. Failure of the simulation to capture the experimentally observed disappearance of the *H, *I and *K elements may originate from the misidentification

of some sites in these regions as recrystallized nuclei and their subsequent disproportionate growth. Close inspection of KAM values (Fig. 3) shows isolated points with KAM > 9 occasionally occur in regions of otherwise low deformation where twins of different modes or variants cross each other, and are likely the primary source of such spurious nucleation sites. Replacing the constant recrystallization boundary velocity with a version scaled by the stored energy released, as discussed in the previous section, could help suppress the contribution of the larger grains in the current simulation, as their nuclei are located within regions of low deformation where the recrystallized grains would expand more slowly. A 3D initial dataset could also rectify the problem by providing nucleation sites within material above and below the present section plane.

In the absence of the larger grains, the remainder of the simulated microstructure (Fig. 10, Grains < 300 μm^2) is close to the experimental recrystallization texture. The only remaining difference is the predominant *C and *G elements of the texture remain approximately 60% more intense than expected. Texture strength can be expressed quantitatively as a texture index (J_{ODF}) that will take the value of 1 at complete randomization, and infinity for a perfect single crystal [71]. The deformed microstructure (Fig. 1) has a texture index of 5.9, which weakens to a recrystallized state of 2.6 (Fig. 6). For comparison the HAGB: KAM > 9 texture (Fig. 9) has a value of 4.3, and the Grains < 300 μm^2 subset has a value of 4.1. None of the nucleation conditions tested were able to match the degree of randomization observed experimentally. This is because the simulation currently selects recrystallized grain orientations from a sub-set of the orientations present in the deformed microstructure. As there are few grains oriented significantly away from ND in the (001) pole figure in the deformed microstructure, it is impossible for those orientations to appear during the simulation. One possibility is that the

EBSD map was not acquired at a sufficient resolution to capture all of the possible sub-grain orientations that would later become nuclei, and that it systematically failed to identify sub-grains with more random orientations.

Another possibility is that the deformation microstructure in Fig. 1 has not completed recovery and that the dislocation walls that separate sub-grains are still forming. In order to acquire a well-indexed EBSD pattern, the deformation microstructure was annealed at 450°C for 30 seconds prior to analysis [31], which was believed to be sufficient for dislocations to form a recovery microstructure and establish sub-grain boundaries. Due to the complexity of deformation modes in α-uranium, and the large intra-granular misorientations still present in Fig. 1, this assumption may be incorrect. If dislocations continue to rearrange prior to recrystallization, a broader spread of site orientations than observed in Fig. 1 could be formed. The deformation modes active in α-uranium are complex and highly dependent on temperature and nature of the applied stress [5 - 10]. Warm clock-rolling at 300°C is expected to result in high activity of the [100](010), [100](001) and $\frac{1}{2}\langle 1\bar{1}0\rangle\{110\}$ slip modes, in addition to deformation twinning. A tilt wall sub-grain boundary formed by [100](001) “floor” mode dislocations will rotate the c-axis of the material forming a sub-grain in the c-a plane about the b-axis. For the texture of clock-rolled α-uranium, with a strong alignment of the (010) axis along RD, evolving floor mode tilt boundaries would therefore lead to the experimentally observed spread of *G elements in the (001) pole figure along TD. Tilt walls of the [100](010) “wall” and $\frac{1}{2}\langle 1\bar{1}0\rangle\{110\}$ “chimney” modes would both result in analogous rotations about the c-axis and spreading of the *C element away from RD. Through these mechanisms it possible that the nucleation sites identified in the model would experience rotation away from the strong *C and *G features before the onset recrystallization as their sub-grain boundaries form, explaining the

observed differences between the experimental recrystallization texture (Fig. 6) and the simulated recrystallization texture (Fig. 10, Grains < 300 μm^2). It is important to note that this potential mechanism, involving the formation of sub-grains during recovery and their rotations from the deformed parent grain matrix, is distinct from continuous sub-grain rotation during dynamic recrystallization. None of the possible explanations for why the model fails to capture the randomization is exclusive and further work exploring the effects of EBSD scan resolution and recovery times will be needed to sort out their relative contributions.

Recrystallization textures in α-uranium are expected to be temperature dependent, based upon prior results obtained on material which was cold straight-rolled (at room temperature) and recrystallized using a range of different heat treatments (Fig. 11) [20]. As a consequence, care must be taken when extrapolating some results presented in this paper to other temperature ranges. The deformation texture of cold straight-rolled α-uranium is relatively close to the warm clock-rolled texture, with strong alignment of (010) along RD, a four-fold *A element in (100), and a split TD *G element in (001). Where the warm clock-rolled deformation texture has equally strong intensity along elements *H-*K, the cold straight-rolled texture has a very prominent *H element, moderate *K, and notably weaker *I and *J elements. Recrystallization at 380°C (300 hr) results in only minor evolution in the texture, primarily weakening of the *I and *J elements and appearance of the *B element. In stark contrast, recrystallization at 450°C (15 hr) results in the complete disappearance of the strong *H element along with *I and *J, in a comparable trend to the warm clock-rolled recrystallization texture at 450°C. To ensure the difference observed between 380°C and 450°C did not result from texture evolution during grain growth, samples recrystallized at 380°C were subsequently subjected to a 450°C grain growth treatment, producing a stronger *B element but few other changes. There are at least two

possible explanations for the temperature dependency of the α-uranium recrystallization texture. Recrystallization at lower temperatures over longer times is likely to result in a greater density of nuclei forming within the microstructure, which likely means nucleation on HAGB sites with lower stored energies. Additionally, the strong temperature dependence of several hard slip systems in α-uranium, e.g. the $\frac{1}{2}\langle 1\bar{1}0\rangle\{110\}$ chimney mode, means that the recovery microstructure may vary significantly with temperature and alter the orientation of sub-grains that go on to form nuclei. If the contribution from the number density of nuclei is dominant, lower temperature recrystallization textures of warm clock-rolled samples should resemble the HAGB: KAM > 6 or KAM > 7 conditions, producing a stronger *A texture and weaker *B. Conversely, if sub-grain boundary evolution plays a large role, a texture that does not fit the KAM threshold based trend should be observed. Recrystallization of warm rolled α-uranium at temperatures at 550°C [7] produces a similar texture to 450°C, so recrystallization experiments at lower temperatures and longer times are necessary to establish the temperature dependency on crystallographic texture.

As crystal plasticity computational techniques have started to successfully predict α-uranium deformation microstructures [11, 12, 72] , one of the remaining hurdles to simulating the full thermomechanical history of uranium components is developing an improved understanding of nucleation processes within the material. It is here that Potts modeling used in conjunction with experimental EBSD data is able to provide a key contribution by identifying conditions within the microstructure at different temperatures and deformation conditions that are most likely to form recrystallization nuclei. These conditions can then be incorporated into microstructural simulations to extend them through recrystallization processes and predict final microstructures, similar to Radhakrishnan et al. [41]. Thus, the application of Potts MC

recrystallization models to EBSD data to screen for preferential nucleation conditions within the microstructure also has potential to help resolve issues pertaining to the origination of the so called “rare-earth” texture in Mg-RE, Mg-RE-Zn, and Mg-Ca-Zn alloys, which currently remains unresolved [73].

6. Conclusions

A Monte Carlo Potts model was used to simulate static recrystallization textures resulting from annealing of warm-rolled α-uranium. In particular, different nucleation site selection concepts applied to experimental EBSD data revealed that nuclei form, as previously hypothesized, along non-twin HAGBs in the deformed microstructure at 450°C. The nuclei do not occur homogenously along the HAGBs, however, and are preferentially located in areas of large stored energy which are indicated by especially high intra-granular misorientations measured using the Kernel Average Misorientation (KAM) value. In the future, trends noted between the location of the recrystallized nuclei within the deformed microstructure and the resulting simulated textures can in turn be applied to studies on the temperature dependency of α-uranium recrystallization textures. Pinpointing the microstructural conditions that can be used to best identify nuclei within α-uranium can inform how deformation microstructures simulated using crystal plasticity techniques will respond to recrystallization.

Acknowledgements

Funding for this research was provided by the Y 12 National Security Complex under the Plant Directed Research, Development, and Demonstration program. This work of authorship and those incorporated herein were prepared by Consolidated Nuclear Security, LLC (CNS) as accounts of work sponsored by an agency of the United States Government under Contract DE NA 0001942. Neither the United States Government nor any agency thereof, nor CNS, nor any

 Work performed at Los Alamos National Laboratory is supported by the Los Alamos National Laboratory Directed Research and Development (LDRD) project 20140630ER. Los Alamos National Laboratory is operated by Los Alamos National Security LLC under DOE Contract DE-AC52-06NA25396. Electron microscopy was performed at the Los Alamos Electron Microscopy Laboratory.

References

[1] J. E. Burke, A. M. Turkalo, The growth of Uranium upon thermal cycling, Transactions of the ASM 50 (1957), 943–953.

[2] F. Foote, Physical metallurgy of uranium, in Progress in NuclearEnergy , Series V, Metallurgy and Fuels, New York, McGraw-Hill, 1956, 81–201.

[3] R. Anderson, D. Carpenter, T. Kollie, D. Smith, Texture-property relationships in the uranium-2.4 weight percent niobium alloy, in Metallurgical Technology of Uranium and Uranium Alloys, Metals Park, OH, American Society for Metals, 1982, 231–249.

[4] O. Sherby, D. Bly, D. Wood, Plastic flow and strength of uranium and its alloys, in Physical Metallurgy of Uranium Alloys, Chestnut Hill, MA, Brook Hill Publishing Co, 1976, 311–348.

[5] D. Brown, M. Bourke, B. Clausen, R. Korzekwa, R. McCabe, T. Sisneros, D. Teter, Temperature and direction dependence of internal strain and texture evolution during deformation of uranium, Mater. Sci. Eng. A 512 (2009), 67–75.

[6] R. McCabe, L. Capolungo, Marshall, C. Cady, C. Tomé, Deformation of wrought uranium: Experiments and modeling, Acta Materialia 58 (2010), 5447–5459

[7] M. Knezevic, L. Capalungo, C. Tome, R. Lebensohn, D. Alexander, B. Mihalia, R. McCabe, Anisotropic stress–strain response and microstructure evolution of textured α-uranium, Acta Materialia 60 (2012), 702–705.

[8] M. Knezevic, R. McCabe, R. Lebensohn, C. Tome, C. Liu, M. Lovato, B. Mihaila, Integration of self-consistent polycrystal plasticity with dislocation density based hardening laws within an implicit finite element framework: Application to low-symmetry metals, J. Mech. Phys. Solids 61 (2013), 2034–2046.

[9] M. Knezevic, R. McCabe, C. Tome, R. Lebensohn, S. Chen, C. Cady, G. G. III, B. Mihaila, Modeling mechanical response and texture evolution of α-uranium as a function of strain rate and temperature using polycrystal plasticity, Int. J. Plast. 43 (2013), 70–84.

[10] C. Calhoun, E. Garlea, R. Mulay, T. Sisneros, S. Agnew, Investigation of the effect of thermal residual stresses on deformation of α-uranium through neutron diffraction measurements and crystal plasticity modeling, Acta Materialia, 85 (2015), 168–179.

[11] M. Zecevic, M. Knezevic, I. J. Beyerlein and R. J. McCabe, Origin of texture development in orthorhombic uranium, Materials Science & Engineering A 665 (2016), 108–124.

[12] M. Zecevic, M. Knezevic, I. J. Beyerlein, R. J. McCabe, Texture formation in orthorhombic alpha-uranium under simple compression and rolling to high strains, Journal of Nuclear Materials, 473 (2016), 143–156.

[13] R. Doherty, D. Hughes, F. Humphreys, J. Jonas, D. Jensen, M. Kassner, W. King, T. McNelley, H. McQueen, A. Rollett, Current issues in recrystallization: a review, Materials Science and Engineering A 238 (1997), 219–274.

[14] R. Doherty, Recrystallization of Metallic Materials, Verlag: Berlin, 1978.

[15] F. Humphreys, M. Hatherly, Recrystallization and Related Annealing Phenomena, Oxford: Pergamon Press, 1995.

[16] R. D. Doherty, Recrystallization and texture, Progress in Materials Science 42 (1997), 39–58.

[17] A. Rollett, Overview of modeling and simulation of recrystallization, Progress in Materials Science 42 (1997), 79–99.

[18] R. Cahn, Plastic deformation of alpha-uranium; twinning and slip, Acta Materialia 1 (1953), 49-70.

[19] M. Yoo, Slip modes of alpha uranium, Journal of Nuclear Materials, 26 (1968), 307-318.

[20] L. T. Lloyd, M. H. Mueller, Recrystallization in rolled uranium sheet, ANL-63Z7, Argonne National Laboratory, Argonne, Illinois, 1962.

[21] M. Mueller, H. Knott, Preferred orientation in rolled and in recrystallized high-purity uranium, ANL-5887, Argonne National Laboratory, Lemont, Illinois, 1959.

[22] E. C. Miller, W. H. Bridges, Metallurgy Division Quarterly Progress Report for Period Ending July 31,1951, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1952.

[23] N. D. Libanati, D. Calais and Lacombe, Recristallisation secondaire et recristallisation apres ecrouissage critique de l'uranium de haute purete, Journal of Nuclear Materials 10 (1963), 23-42.

[24] W. R. McDonell, Preferred orientation of cold-rolled and recrystallized uranium plate, DP - 258, Savannah River Laboratory, Aiken, South Carolina, 1957.

[25] G. Y. Sergeev , V. T. Kolonneva, Recrystallization of Cold-Rolled Uranium, Journal of Nuclear Energy Parts A/B Reactor Science and Technology, 17 (1963), 444–447.

[26] P. Madsen, The behavior of interfaces in lightly worked uranium during recrystallization, Journal of the Institute of Metals 85 (1956), 71–75.

[27] W. Chernock, A. Beck, Quantitative determination of rolling and recrystallization textures in 600C and 300C rolled uranium rods, U.S. Atomic Energy Commission, Technical Information Service, Oak Ridge, TN, 1955.

[28] J. Bingert, R. J. Hanrahan, R. Field, Dickerson, Microtextural investigation of hydrided alpha-uranium, J. Alloys Compounds 365 (2004), 138–148.

[29] D. Carpenter and J. Bullock, OIM and EDX determination of the orientation dependence of corrosion in uranium metal, Inst. Physics Conf. Series 165 (2000), 209–210.

[30] R. McCabe and D. Teter, Analysis of recrystallized volume fractions in uranium using electron backscatter diffraction, Journal of Microscopy 223 (2006), 33–39.

[31] R. McCabe, A. Richards, D. Coughlin, K. Clarke, I. Beyerlein, M. Knezevic, Microstructure effects on the recrystallization of low-symmetry alpha-uranium, Journal of Nuclear Materials 465 (2015), 189-195.

[32] H. Hallberg, Approaches to Modeling of Recrystallization, Metals, 1 (2011), 16–48.

[33] K.G.F. Janssens, D. Raabe, E. Kozeschnik, M. Miodownik, B. Nestler, Computational Materials Engineering, Elsevier Academic Press, 2007

[34] M. Miodownik, A review of microstructural computer models used to simulate grain growth and recrystallisation in aluminium alloys, Journal of Light Metals, 2 (2002), 125–135

[35] M. Miodownik, Monte carlo Potts model, in Computational Materials Engineering, Elsevier Academic Press, 2007.

[36] S. Esche, Monte carlo simulations of grain growth in metals, in Applications of Monte Carlo Method in Science and Engineering, InTech, 2011.

[37] F. Wu, The Potts Model, Rev. Mod. Phys. 54 (1982), 235.

[38] E. Holm, M. Miodownik, A. Rollett, On abnormal subgrain growth and the origin of recrystallization nuclei, Acta Mater. 52 (2003), 2701.

[39] D. Solas, C. Tome, O. Engler and H. Wenk, Deformation and recrystallization of hexagonal metals: Modeling and experimental results for zinc, Acta Materialia 49 (2001), 3791–3801.

[40] Y. Chun, S. Semiatin and S. Hwang, Monte Carlo modeling of microstructure evolution during the static recrystallization of cold-rolled, commercial-purity titanium, Acta Materialia 54 (2006), 3673–3689.

[41] B. Radhakrishnan, S. G. B. and T. Zacharia, Modeling the kinetics and microstructural evolution during static recrystallization—Monte Carlo simulation of recrystallization, Acta Materialia 46 (1998), 4415–4433.

[42] W. Read, W. Shockley, Dislocation models of crystal grain boundaries, Phys. Rev. 78 (1950), 275.

[43] W. Shockley and W. Read, Quantitative predictions from dislocation models of crystal grain boundaries, Phys. Rev. 75 (1949), 692.

[44] R. A. Lebensohn, N-site modeling of a 3D viscoplastic polycrystal using fast Fourier transform, Acta Materialia 49 (2001), 2723–2737.

[45] E. Woldt, D. Juul Jensen, Recrystallization kinetics in copper:comparison between techniques, Metall. Mater. Trans. A 26 (1995), 1717–1724.

[46] M. Black, R. Higginson, An investigation into the use of elec-tron back scattered diffraction to measure recrystallised fraction, Scripta Mater 41 (1999), 125–129.

[47] J. Tarasiuk, Gerber, B. Bacroix, Estimation of recrystallized volume fraction from EBSD data, Acta Materialia 50 (2002), 1467–1477.

[48] F. Caleyo, T. Baudin, R. Penelle, Study of the development of thecube texture in Fe-50% Ni during recrystallization and normal grain growth, Eur. Phys. J. Appl. Phys 20 (2002), 77–89.

[49] D. Dingley, Progressive steps in the development of electron back-scatter diffraction and orientation imaging microscopy, J. Microsc. 213 (2004), 214–224.

[50] W. Wang, A. Helbert, F. Brisset, M. Mathon, T. Baudin, Monte Carlo simulation of primary recrystallization, Acta Materialia 81 (2014), 457–468.

[51] M. Calcagnotto, D. Ponge, E. Demir, D. Raabe, Orientation gradients and geometrically necessary dislocations in ultrafine grained dual-phase steels studied by 2D and 3D EBSD, Materials Science and Engineering: A 527 (2010), 2738–2746.

[52] D. Field, Trivedi, S. Wright, M. Kumar, Analysis of local orientation gradients in deformed single crystals, Ultramicroscopy 103 (2005), 33–39.

[53] W. Pantleon, Resolving the geometrically necessary dislocation content by conventional electron backscattering diffraction, Scripta Materialia 58 (2008), 994–997.

[54] B. El-Dasher, B. Adams, A. Rollett, Viewpoint: experimental recovery of geometrically necessary dislocation density in polycrystals, Scripta Materialia 48 (2003), 141–145.

[55] C. Dahlberg, Y. Saito, M. Öztop, J. Kysar, Geometrically necessary dislocation density measurements associated with different angles of indentations, International Journal of Plasticity 54 (2014), 81–95.

[56] A. Godfrey, W. Q. Cao, Q. Liu, N. Hansen, Stored energy, microstructure, and flow stress of deformed metals, Metallurgical and Materials Transactions A, 36 (2005), 2371–2378.

[57] D.P. Field, C.C. Merriman, N. Allain-Bonasso, F. Wagner, Quantification of dislocation structure heterogeneity in deformed polycrystals by EBSD, Modelling Simul. Mater. Sci. Eng., 20 (2012), 024007.

[58] J.K. Mason, J. Lind, S.F. Li, B.W. Reed, M. Kumar, Kinetics and anisotropy of the Monte Carlo model of grain growth, Acta Materialia, 82 (2015), 155–166

[59] J.K. Mason, Grain boundary energy and curvature in Monte Carlo and cellular automata simulations of grain boundary motion, Acta Materialia, 94 (2015), 162–171

[60] P.E. Goins, E.A. Holm, The Material Point Monte Carlo model: A discrete, off-lattice method for microstructural evolution simulations, Computational Materials Science, 124 (2016), 411–419

[61] D. Raabe, Scaling Monte Carlo Kinetics of the Potts Model using Rate Theory, Acta Materialia, 48 (2000), 1617–1628

[62] M. Mueller, H. Knott, Beck, Deformation and recrystallization textures of rolled uranium sheet, J. Metals 203 (1955), 1214–1218.

[63] C. Choi and M. Staker, Neutron diffraction texture study ofdeformed uranium plates, J. Mater. Sci . 31 (1996), 3397–3402.

[64] J. Morrel and M. Jackson, Uranium Processing and Properties, Springer-Verlag, New York, 2013.

[65] F. Bachmann, R. Hielscher, H. Schaeben, Texture analysis with MTEX - free and open source software toolbox, Solid State Phenomena 160 (2010), 63.

[66] M. Steiner, J. Bhattacharyya and S. Agnew, The Origin and Enhancement of $\{0001\}\langle 11\bar{2}0\rangle$ Texture during Heat Treatment of Rolled AZ31B Magnesium Alloys, Acta Materialia, vol. 95, 443–455, 2015.

[67] A. Beck, R. Sperry, Strain Induced Grain Boundary Migration in High Purity Aluminum, Journal of Applied Physics 21 (1950), 150.

[68] G. Gottstein, L. S. Shvindlerman, Grain Boundary Migration in Metals: Thermodynamics, Kinetics, Applications, Second Edition, CRC Press, 2009.

[69] E.S. Fisher, Preparation of Alpha Uranium Single Crystals By a Grain-Coarsening Method, Journal of Metals, 9 (1957), 882-888

[70] Marshall, G. Proust, J. Rogers and R. McCabe, Automatic twin statistics from electron backscattered diffraction data, Journal of Microscopy 238 (2010), 218–229.

[71] D. Mainprice, F. Bachman, R. Hielscher, H. Schaeben, Descriptive tools for the analysis of texture projects with large datasets using MTEX: strength, symmetry and components, Special Publications of the Geological Society London 409 (2014), 251–271.

[72] M. Knezevic, B. Drach, M. Ardeljan, I.J. Beyerlein, Three dimensional predictions of grain scale plasticity and grain boundaries using crystal plasticity finite element models, Computer Methods in Applied Mechanics and Engineering, 277 (2014), 239–259

[73] Z. Zeng, Y. Zhu, S. Xu, M. Bian, J. Nie, Texture evolution during static recrystallization of cold-rolled magnesium alloys, Acta Materialia, 105 (2016), 479–494.

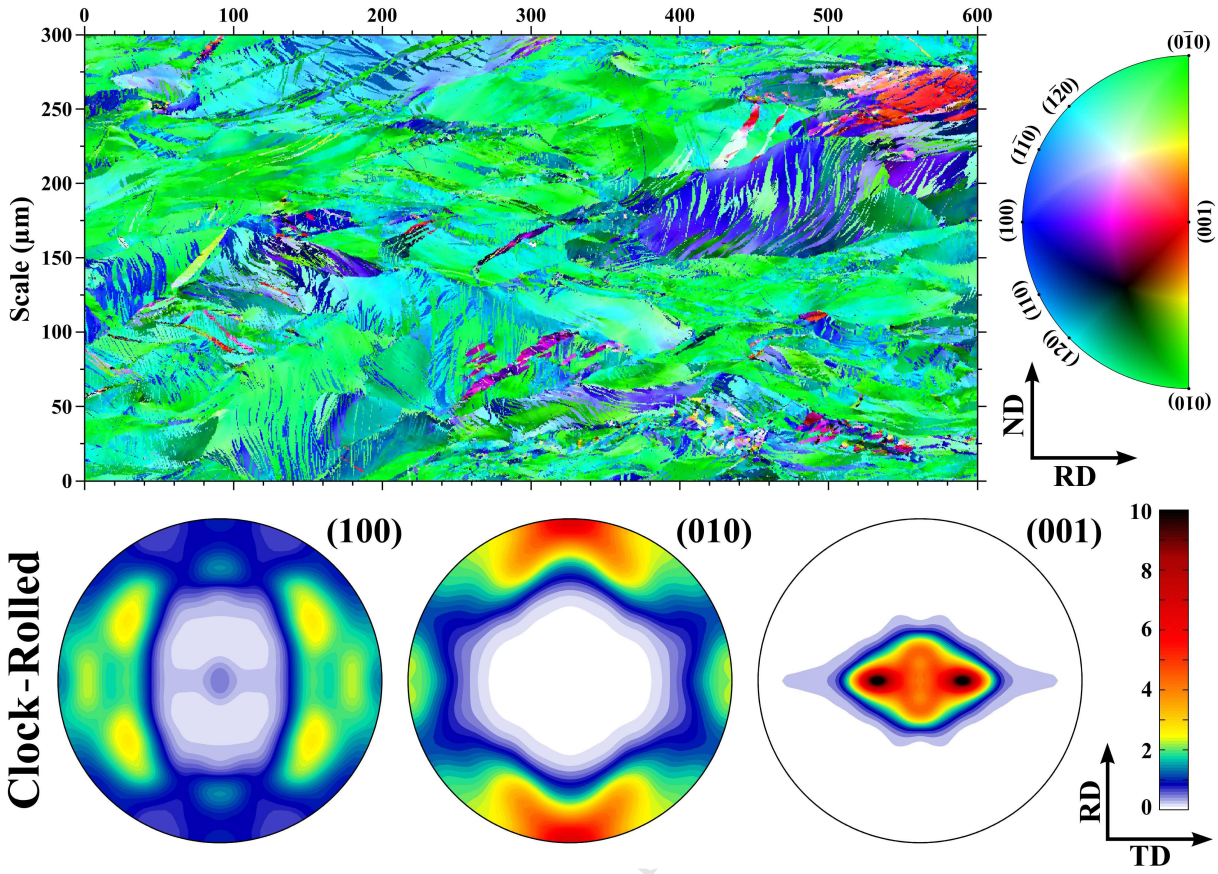

Fig. 1 Experimental EBSD map and stereographic pole figures of the α-uranium warm clock-rolled deformed microstructure adapted from data originally presented in Ref. [31], with inverse pole-figure (IPF) coloring along the RD direction. Inversion symmetry can reduce the IPF map further by half, but has not been applied to maintain contrast in the figure. The microstructure is dominated by multiple modes of deformation twinning and large intragranular misorientation gradients. There is a strong alignment of the (010) axes along the rolling direction, and the (001) axes along ND with a clear splitting along TD.

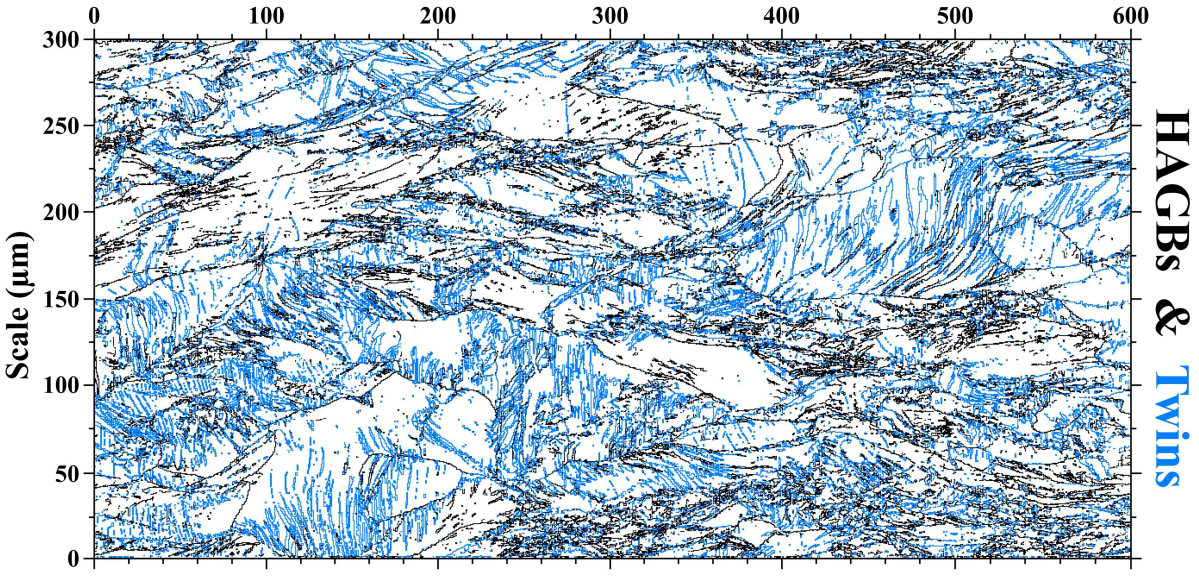

Fig. 2 Map of grain boundaries designated either as HAGBs or Twins for the α-uranium warm clock-rolled deformed microstructure seen in Fig. 1.

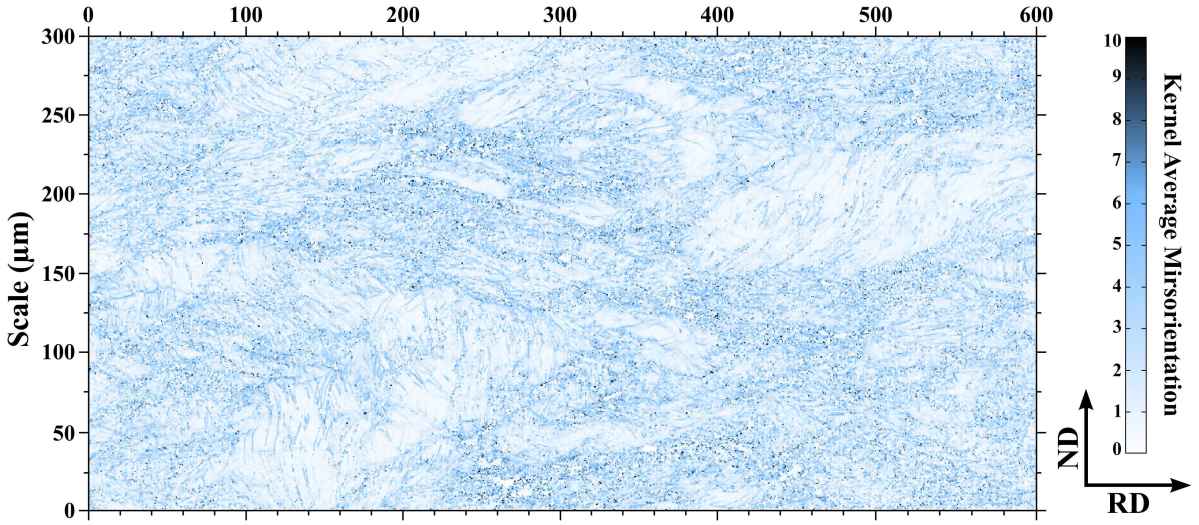

Fig. 3 Map of KAM values (3x3 grid, 10° threshold) for the α-uranium warm clock-rolled deformed microstructure seen in Fig. 1, showing heterogeneity between different regions.

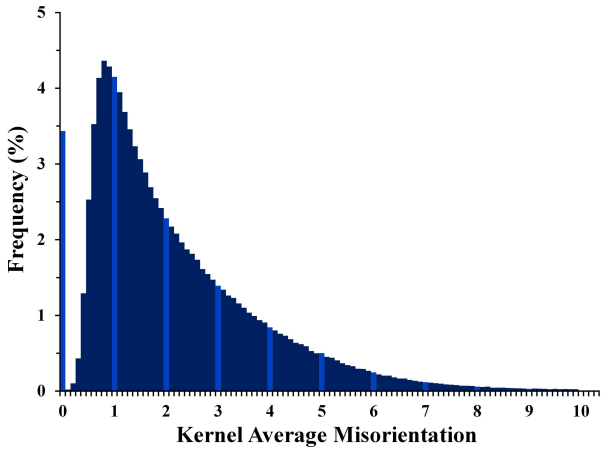

Fig. 4 Histogram of KAM values (3x3 grid, 10° threshold) for the α-uranium warm clock-rolled deformed microstructure seen in Fig. 1, exhibiting an approximately log-normal distribution.

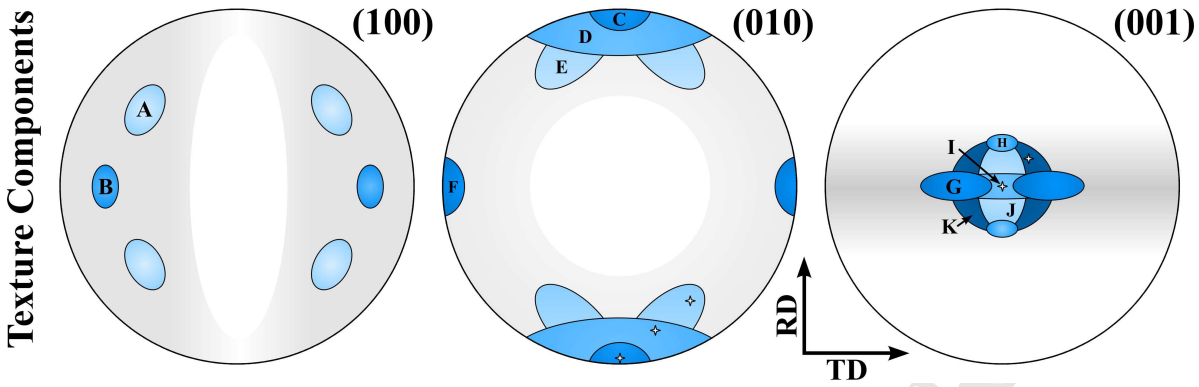

Fig. 5 Graphical legend of distinct texture elements (*) that appear repeatedly within the deformed and recrystallized α-uranium textures. The small crosses (+) in several of the elements mark representative locations used later in Fig. 8.

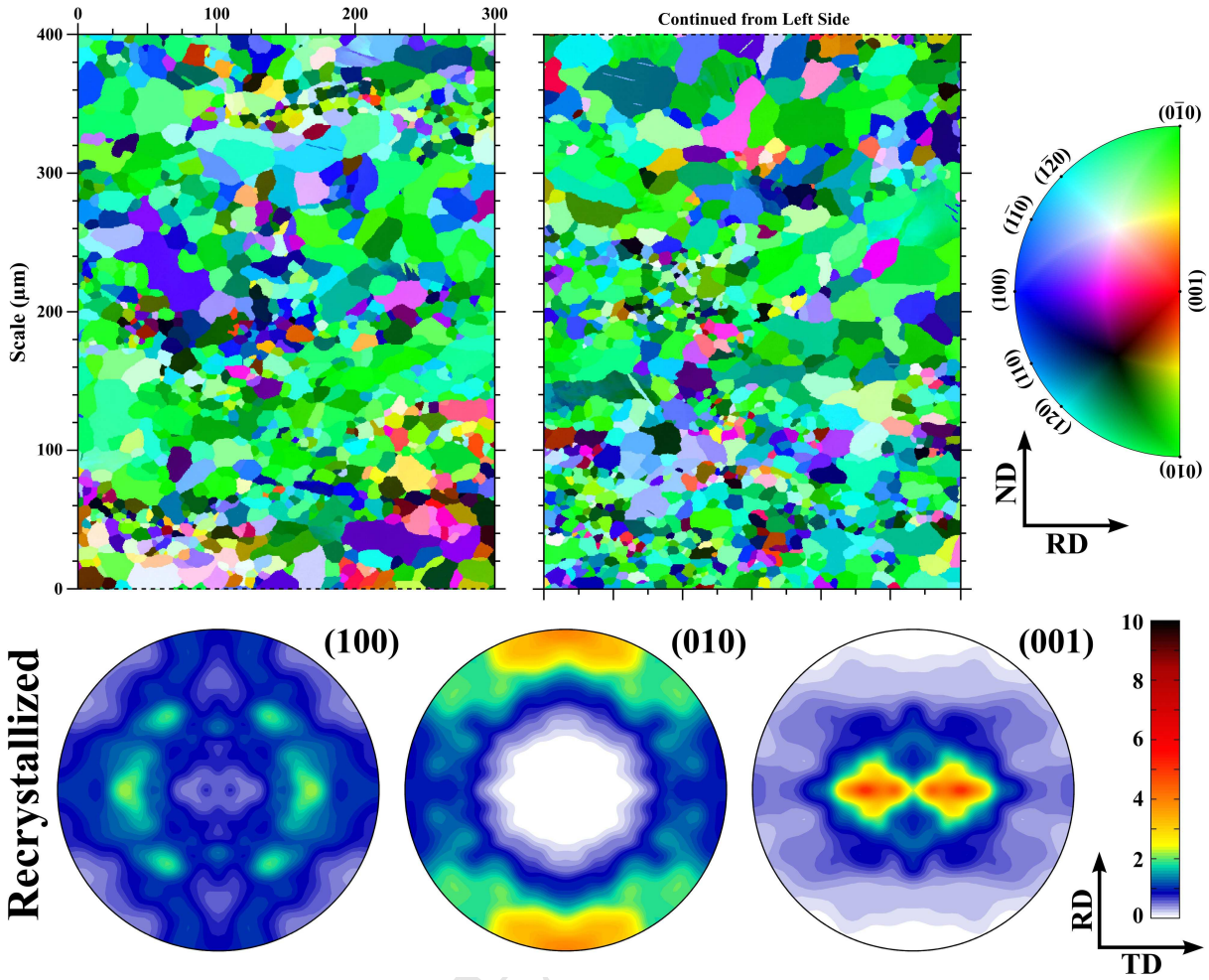

Fig. 6 Experimental EBSD map and stereographic pole figures of the α-uranium warm clock-rolled deformed microstructure recrystallized at 450°C for 10^5 seconds, originally presented in Ref. [31]. Inverse pole-figure (IPF) coloring is along the RD direction. The recrystallized grains are clustered into like-sized bands across the microstructure and the texture has evolved from the deformed state, becoming notably weaker.

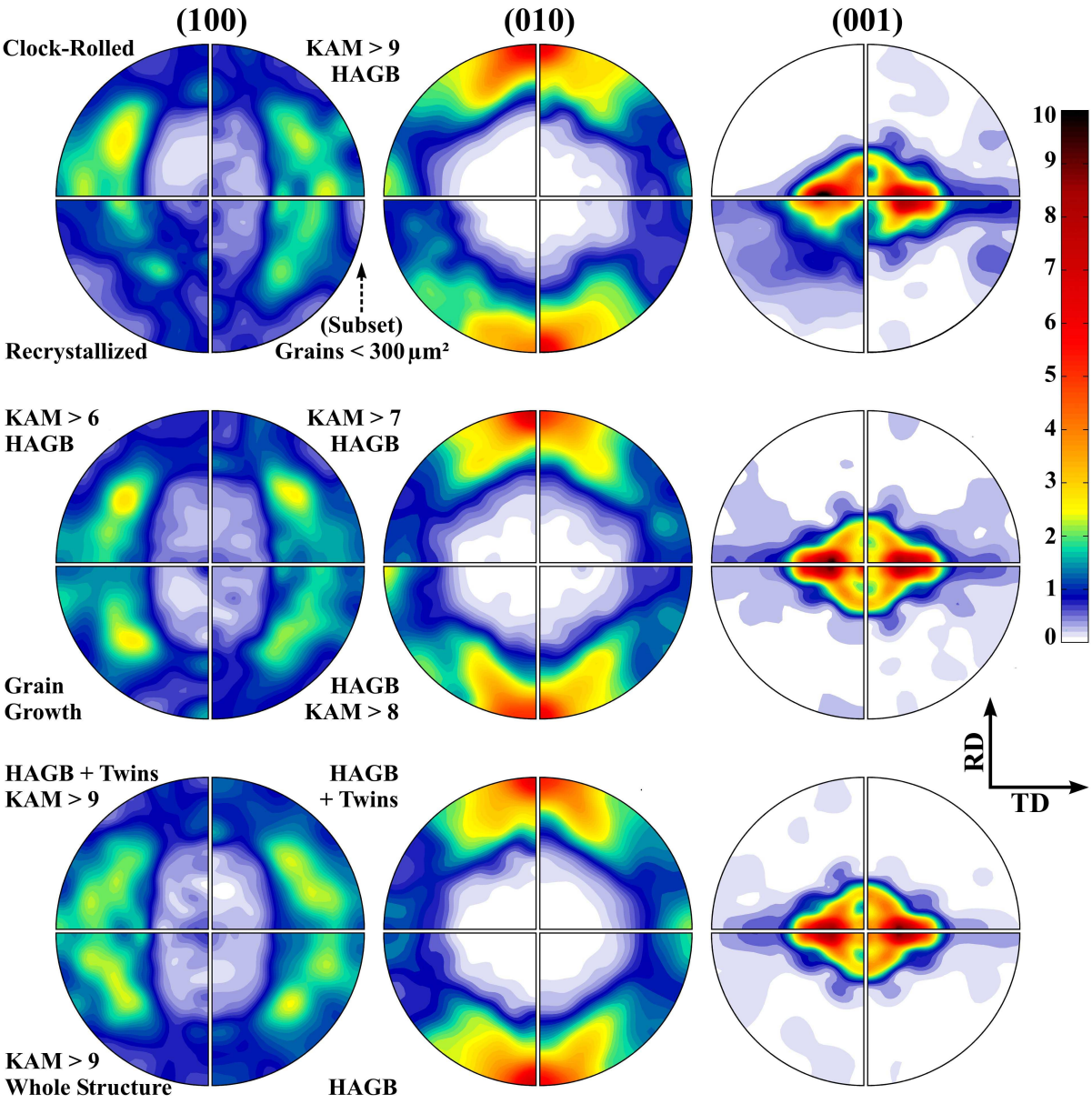

Fig. 7 Simulated EBSD stereographic pole figures for a number of recrystallized nuclei conditions, along with the experimental clock-rolled and recrystallized textures. The pole figures have been reduced to show only a fundamental quadrant, applying the rolled sample's orthotropic symmetry.

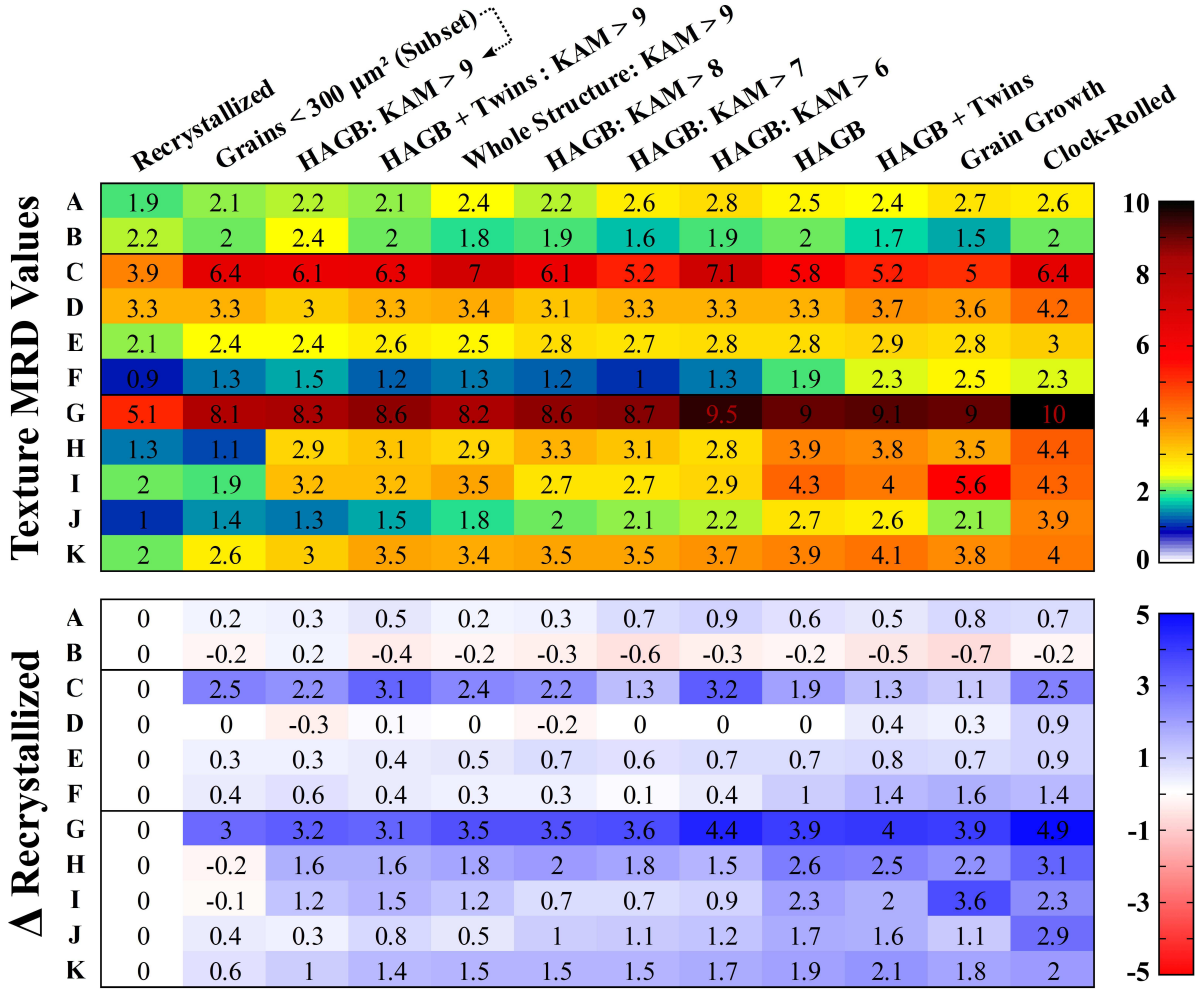

Fig. 8 Trends in the strength of the different elements (Fig. 5) between both experimentally determined and simulated recrystallization textures found in Fig. 7. Note: The importance of each element is not equal and the values do not always capture changes to the spread or shape of the elements, which are better observed in their respective pole figures.

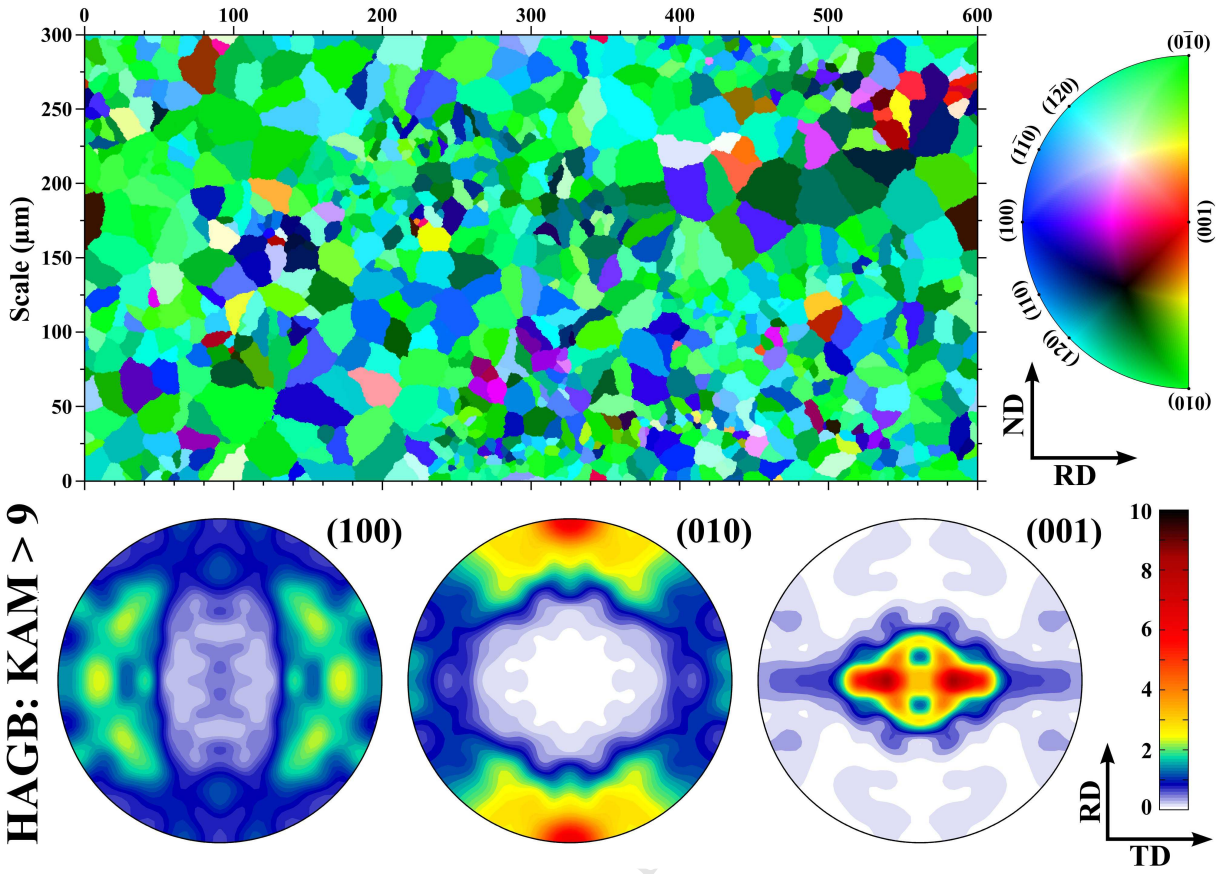

Fig. 9 Simulated EBSD map and stereographic pole figures of a recrystallized α-uranium warm clock-rolled microstructure, with inverse pole-figure (IPF) coloring along the RD direction. This figure was generated starting from the microstructure in Fig. 1 and employing the Potts MC recrystallization model, with recrystallized nuclei set as all points along non-twin HAGBs with KAM > 9.

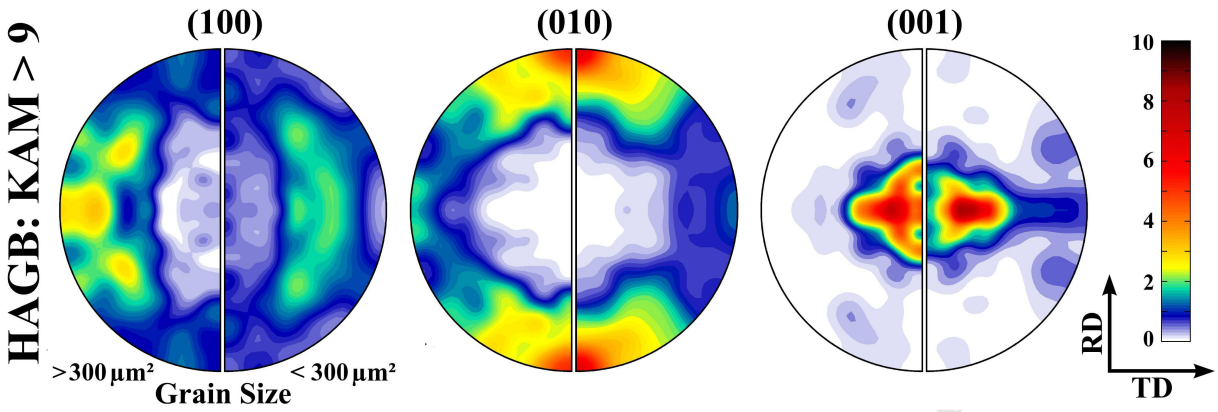

Fig. 10 Simulated EBSD stereographic pole figures for the recrystallized α-uranium warm clock-rolled texture in Fig. 9 (HAGB: KAM > 9) broken out into sub-textures for Grains > 300 μm^2 and Grains < 300 μm^2.

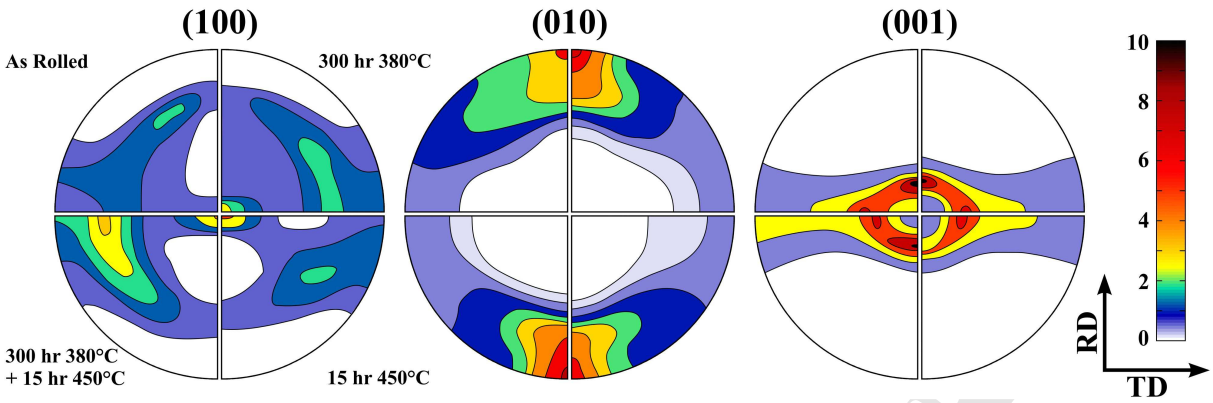

Fig. 11 Experimental XRD stereographic pole figures of straight cold-rolled α-uranium and recrystallization textures at different heat treatment conditions, adapted from raw data presented in Ref. [20] and converted into approximate MRD scale values for comparison.

Highlights:

- Monte Carlo Potts modelling and experimental electron backscattered diffraction data are used to simulate the static recrystallization texture that results from annealing warm-rolled α-uranium.
- This study demonstrates that, consistent with prior observations, recrystallized nuclei in α-uranium form preferentially on non-twin high-angle grain boundary sites at 450°C.
- In a new finding, nucleation along these boundaries is found to occur only at a highly constrained subset of sites possessing the largest degrees of local deformation.
- These nucleation conditions can be incorporated into *ab initio* microstructural simulations to extend them through recrystallization processes and predict final microstructures.

Instructions for the RXMonteCarlo2D C++ Code

This code was written at the University of Virginia by Matt Steiner, Ph.D. in conjunction with Professor Sean Agnew, Ph.D. It is being provided as open source material, and the authors ask that those wishing to use or build upon the code cite the paper it was originally distributed with:

M.A. Steiner, R.J. McCabe, E. Garlea and S.A. Agnew, "Monte Carlo Modeling of Recrystallization Processes in alpha-Uranium", Journal of Nuclear Materials (2017)

This code represents an expansion of the grain growth code originally distributed with:

M.A. Steiner, J.J. Bhattacharyya, and S.R. Agnew, "The Origin and Enhancement of {0001}<11-20> Texture during Heat Treatment of Rolled AZ31B Magnesium Alloys", Acta Materialia (2015)

And now includes recrystallization and recovery phenomena. These papers describe the functionality of the code and will be a helpful starting point for those wishing to modify it. Newer versions of the code may be available and we encourage those interested to contact the authors.

In addition to the obvious expansions in the code several changes have been made in the internal structure of the program between versions. Energy and mobility terms are now calculated to the nearest 1 degree increments, saving a great deal of computational speed. Grain orientation data now has all three Euler angles rounded into integer steps at the beginning with a user set interval, which was necessary for recrystallization cases as each grain no longer has one average value and computation costs are raised.

The RXMonteCarlo2D program supports all possible crystal symmetry groups. As distributed the code currently only supports input and output in Oxford Instruments' HKL Channel 5 .CTF format. The object oriented implementation, however, should make expansion to other file formats straightforward. The program does not currently support any common speedup algorithms (e.g. tracking boundary sites) as the 2D implementation keeps operation time fairly manageable. A Read-Shockley grain boundary energy and mobility are currently hard coded into program. KAM values are read into the program in the standard export format of Oxford instruments, one column header line followed by four columns containing: index, X, Y, KAM.

The code is commented throughout to assist the programmer but a following overview of the class structure may prove helpful:

RXMonteCarlo2D :
Core class of the program including the main() call and all parameters that will need to be changed regularly, including the files to load and print to, symmetry, the number of time steps, and the cyclic number of time steps to print after

FileHandler :
Responsible for loading information from a .CTF EBSD data file as well as printing back in this format. The class that would need to be expanded if wanting to add additional file support.

MonteCarloEngine:
Handles the core of the Monte Carlo simulation. A number of different nucleation events and time step functions are available, with comments describing how each operates.

GrainMap :
Holds a matrix of the grain identifications of all pixel positions , as well as a list of all the grains in the microstructure

Grain:
A grain object in the program. Has a unique identification number, set of three Euler angles, and a rotation matrix relative to the reference frame.

MisorientationAngles:
Computes and stores the misorientation angle between all grains in the microstructure using proper symmetry. The most computationally intensive step in the program. By computing and storing this data only once before starting the MC simulation significant computational savings are made for most simulations, even if many of the grains will never border. The handling of this class could be optimized significantly if the matrix were initialized the first time each calculation is attempted rather than upfront. If the program is taking longer than a minute or two to pass through the initiation of this class then implementing such an optimization may be worthwhile to the programmer.

BoundaryProperties:
Computes and stores the mobilities and energies between misorientation angles based upon a Read-Shockley model.

Neighbors:
Used to calculate the position of neighboring pixels while imposing periodic boundary conditions

Matrix:
3 x 3 Matrix object, specifically introduced for storing rotation matrices. Can transform three Euler angles in Bunge convetion to a rotation matrix using constructor. May need to add a new constructor if adding an input file that is not in Bunge convention.

```
/*RXMonteCarlo2D.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt


#include <iostream>
#include "FileHandler.h"
#include "MisorientationAngles.h"
#include "BoundaryProperties.h"
#include "GrainMap.h"
#include "MonteCarloEngine.h"

using namespace std;

int main()
{
      //set file location to load, file location basename for saving, and optionally a KAM file to load)
      FileHandler file = FileHandler("/cygdrive/c/Users/You/Folder/filename.CTF",
                  "/cygdrive/c/Users/You/Folder/printfileBaseName",
                  "/cygdrive/c/Users/You/Folder/KAMfilename.txt");


      // From designations in Kocks "Texture and Anisotropy"
      // ---------------------------
      // Hexagonal  6 | 622, 32, 6, 3
      // Cubic      3 | 432, 23, 3
      // Tetragonal 4 | 42, 4
      // Orthogonal 4 | 222
      // Monoclinic 4 | 2
      // Triclinic  4 | 1

      //manually input symmetry of the sample
      int symBranch = 4;
      int symGroup = 222;

      //simulation lattice temperature, recommended 0.6-0.7, prevents hang-ups on 2D square lattice w/8 neighbors
      float symTemp = 0.66;

      //mobility scaling factor (m <= 1) for grain growth portion of the simulation
      float mobility = 0.2;

      //Reads in the file, reduces it down to int type data of reduced order (e.g. Euler angles binned by 5) for
size, minimum of 1
      GrainMap initialMap = file.InitialMap(5);

      //prints the binned map to file for record (optional line)
```

```
file.PrintMapToCTFFile(initialMap,0);

//initiates MC Engine
MonteCarloEngine mcEngine = MonteCarloEngine(initialMap,symTemp,symBranch,symGroup,mobility);


//*************************************************************************************
//  The following code is an example and must be editted. It demonstrates running
//  multiple simulations from the same starting structure, which reuses the intense
//  intialization calculations and saves computation time.
//*************************************************************************************

//enter MC steps to run, number of steps to save a new file after (repeating)
int timeSteps = 200;
int printEvery = 20;

//nucleates recrystallized grains, multiple functions available in MCEngine (see MCEngine.h)
//can be placed inside following loop for multiple instances
mcEngine.NucleateRXbyKAMThreshold(9,1);

//loop that executes a MC time step
for(int count = 1; count < timeSteps+1 ; count ++)
{
      //multiple alternate functions available in MCEngine (see MCEngine.h)
      mcEngine.OneTimeStepRXOnlyHighMobility();

      cout << "Iteration " << count << " run\n";

      //prints an output file
      if(count % printEvery == 0)
      {
            file.PrintMapToCTFFile(mcEngine.CurrentMap(), "run1-", count);
      }
}

//reuses the initial map and calculations for another independent run of the simulation
mcEngine.resetMapTo(initialMap);
mcEngine.NucleateRXbyKAMThreshold(9,1);
for(int count = 1; count < timeSteps+1 ; count ++)
{
      mcEngine.OneTimeStepRXOnlyHighMobility();
      cout << "Iteration " << count << " run\n";
      if(count % printEvery == 0)
      {
            file.PrintMapToCTFFile(mcEngine.CurrentMap(), "run2-", count);
```

```
            }
        }

        //*************************************************************************************
        //  End of example code
        //*************************************************************************************


        //ends main
        return 0;
}
```

```
/* FileHandler.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef FILEHANDLER_H_
#define FILEHANDLER_H_
#include <string>
#include "Grain.h"
#include "GrainMap.h"
#include "MisorientationAngles.h"
using namespace std;

class FileHandler
{
	private:
		string myInputFile;
		string myOutputFileBase;
		string myKAMFile;
		string myFileHeader;
		float myXStep;
		float myYStep;

	public:
		FileHandler(string inputFileName, string outputFileName);
		FileHandler(string inputFileName, string outputFileBaseName, string theKAMFileName);
		GrainMap InitialMap(int myOrder);
		void PrintMapToCTFFile(GrainMap myGrainMap, int addendum);
		void PrintMapToCTFFile(GrainMap myGrainMap, string addon, int addendum);
		void PrintMapToMapFile(GrainMap myGrainMap);
		void PrintMapToGrainIDFile(GrainMap myGrainMap);
		void PrintMapToKAMFile(GrainMap myGrainMap, int addendum);
};

#endif /* FILEHANDLER_H_ */
```

```
/* FileHandler.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "FileHandler.h"
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <sstream>
#include "Grain.h"
#include "GrainMap.h"
#include <vector>
#include <math.h>
#include "MisorientationAngles.h"

using namespace std;

FileHandler::FileHandler(string inputFileName, string outputFileBaseName)
{
	// sets the input file
	myInputFile = inputFileName;

	// sets the output file base name
	myOutputFileBase = outputFileBaseName;

	// defaults, to be set during InitialMap
	myXStep = 1;
	myYStep = 1;
	myFileHeader = "";

	myKAMFile = "";
}

//reads in standard files along with a Kernel Average Misorientation File (of Channel 5 txt export format)
FileHandler::FileHandler(string inputFileName, string outputFileBaseName, string theKAMFileName)
{
	// sets the input file
	myInputFile = inputFileName;

	// sets the output file base name
	myOutputFileBase = outputFileBaseName;

	// sets the KAM input file
	myKAMFile = theKAMFileName;
```

```
	// defaults, to be set during InitialMap
	myXStep = 1;
	myYStep = 1;
	myFileHeader = "";
}

GrainMap FileHandler::InitialMap(int myOrder)
{
	// Initializes cell count variables
	int myXCells;
	int myYCells;

	// starts the input procedure
	string tempString;
	ifstream myFile (myInputFile.c_str());

	// Stores header and initializes cell and step variables from CTF file
	for(int line = 1; line < 16; line++)
	{
		getline(myFile,tempString);

		myFileHeader += tempString + "\n";

		if(line == 5)
		{
			tempString.erase(0,6);
			myXCells = atoi(tempString.c_str());
		};
		if(line == 6)
		{
			tempString.erase(0,6);
			myYCells = atoi(tempString.c_str());
		};
		if(line == 7)
		{
			tempString.erase(0,5);
			myXStep = atof(tempString.c_str());
		};

		if(line == 8)
		{
			tempString.erase(0,5);
			myYStep = atof(tempString.c_str());
		};
	}
```

```
// Creates temporary storage units for each row's data
short int *eulerAng1;
short int *eulerAng2;
short int *eulerAng3;
eulerAng1 = new short int[myXCells*myYCells];
eulerAng2 = new short int[myXCells*myYCells];
eulerAng3 = new short int[myXCells*myYCells];

// Reads through tabular CTF data and stores Euler angles from each line

for(int row = 0; row < myXCells*myYCells; row++)
{
      getline(myFile,tempString);
      istringstream ss(tempString);
      string tempClip;

      //rounds Euler angles to nearest int "order" degrees
      for(int col = 1; col < 9; col++)
      {
            getline(ss, tempClip, '\t');
            if(col == 6)
            {
                  float temp1 = atof(tempClip.c_str());
                  temp1 = roundf(temp1/myOrder)*myOrder;
                  eulerAng1[row] = (short int) temp1;
            };
            if(col == 7)
            {
                  int temp2 = atof(tempClip.c_str());
                  temp2 = roundf(temp2/myOrder)*myOrder;
                  eulerAng2[row] = (short int) temp2;
            };
            if(col == 8)
            {
                  int temp3 = atof(tempClip.c_str());
                  temp3 = roundf(temp3/myOrder)*myOrder;
                  eulerAng3[row] = (short int) temp3;
            };
      }

      //tracker
      cout << "map row " << row << " read\n";
}
myFile.close();
```

```
//Program Status Update
cout << "Input File Read\n";

//initiates KAM value storage
float *theKAMValues;
theKAMValues = new float[myXCells*myYCells];

//KAM file input is in the standard export format of Oxford instruments, one column header line
//followed by four columns containing: index, X, Y, KAM

//runs if a KAM file needs to be read in
if(myKAMFile != "")
{
	//sets second stream for KAM file
	ifstream myFile2 (myKAMFile.c_str());

	//tosses first line (column titles)
	getline(myFile,tempString);

	//reads in each line
	for(int row = 0; row < myXCells*myYCells; row++)
	{
		getline(myFile2,tempString);
		istringstream ss(tempString);
		string tempClip;

		//grabs the KAM value in the last column
		for(int col = 1; col < 5; col++)
		{
			getline(ss, tempClip, '\t');
			if(col == 4)
			{
				float temp1 = atof(tempClip.c_str());

				//Adds a very small residual KAM for accounting purposes
				//Allows differentiation from new nucleated recystallized grains with KAM=0
				theKAMValues[row] = temp1 + 0.001;
			}
		}

		//tracker
		cout << "KAM row " << row << " read\n";
	}
```

```
            myFile2.close();
      }

      //runs if no KAM file, sets all KAMs to 0
      else
      {
            for(int iteration = 0; iteration < myXCells*myYCells; iteration++)
            {
                  theKAMValues[iteration] = 0;
            }
      }

      //returns a GrainMap object initialized from the file data
      return GrainMap(int(myXCells), int(myYCells), eulerAng1, eulerAng2 ,eulerAng3, theKAMValues);
}

void FileHandler::PrintMapToCTFFile(GrainMap myGrainMap, int addendum)
{
      //sets the file name to be printed to
      ostringstream myFileName;
      myFileName << myOutputFileBase << addendum <<".CTF";

      //creates the output file
      ofstream myFile(myFileName.str().c_str());

      //prints stored file header
      myFile << myFileHeader;

      //prints grain map back into CTF format
      for(int y = 0; y < myGrainMap.GetYSize(); y++)
      {
            for(int x = 0; x < myGrainMap.GetXSize(); x++)
            {
                  //prints "Phase" column data
                  myFile << "1\t";

                  //prints X & Y positions with original step adjustment
                  myFile << myXStep*x << "\t";
                  myFile << myYStep*y << "\t";

                  //prints "Bands" and "Error" lines
                  myFile << "0\t0\t";

                  //finds corresponding grain and prints euler angles
                  Grain tempGrain = myGrainMap.GetGrainAtGrid(x,y);
```

```
                myFile << tempGrain.GetEulerAngle1() << "\t";
                myFile << tempGrain.GetEulerAngle2() << "\t";
                myFile << tempGrain.GetEulerAngle3() << "\t";

                //prints "MAD", "BC" and "BS" lines, terminates line
                myFile << "0\t0\t0\r\n";
          }
     }
     myFile.close();
}

void FileHandler::PrintMapToCTFFile(GrainMap myGrainMap, string addon, int addendum)
{
     //sets the file name to be printed to
     ostringstream myFileName;
     myFileName << myOutputFileBase << addon << addendum <<".CTF";

     //creates the output file
     ofstream myFile(myFileName.str().c_str());

     //prints stored file header
     myFile << myFileHeader;

     //prints grain map back into CTF format
     for(int y = 0; y < myGrainMap.GetYSize(); y++)
     {
          for(int x = 0; x < myGrainMap.GetXSize(); x++)
          {
                //prints "Phase" column data
                myFile << "1\t";

                //prints X & Y positions with original step adjustment
                myFile << myXStep*x << "\t";
                myFile << myYStep*y << "\t";

                //prints "Bands" and "Error" lines
                myFile << "0\t0\t";

                //finds corresponding grain and prints euler angles
                Grain tempGrain = myGrainMap.GetGrainAtGrid(x,y);
                myFile << tempGrain.GetEulerAngle1() << "\t";
                myFile << tempGrain.GetEulerAngle2() << "\t";
                myFile << tempGrain.GetEulerAngle3() << "\t";

                //prints "MAD", "BC" and "BS" lines, terminates line
```

```
                myFile << "0\t0\t0\r\n";
            }
        }
        myFile.close();
}

void FileHandler::PrintMapToMapFile(GrainMap myGrainMap)
{
        //sets the file name to be printed to
        ostringstream myFileName;
        myFileName << myOutputFileBase << ".map";

        //creates the output file
        ofstream myFile(myFileName.str().c_str());

        //prints file header
        myFile << myGrainMap.GetXSize() << "\r\n";
        myFile << myGrainMap.GetYSize() << "\r\n";

        //prints grain map into MAP format
        for(int y = 0; y < myGrainMap.GetYSize(); y++)
        {
            for(int x = 0; x < myGrainMap.GetXSize(); x++)
            {
                //prints X & Y positions with original step adjustment
                myFile << myXStep*x << "\t";
                myFile << myYStep*y << "\t";

                //finds corresponding grain and prints grain ID
                Grain tempGrain = myGrainMap.GetGrainAtGrid(x,y);
                myFile << tempGrain.GetID() << "\t";

                //terminates line
                myFile << "\r\n";
            }
        }
        myFile.close();
}

void FileHandler::PrintMapToGrainIDFile(GrainMap myGrainMap)
{
        //sets the file name to be printed to
        ostringstream myFileName;
        myFileName << myOutputFileBase << ".gid";
```

```
	//creates the output file
	ofstream myFile(myFileName.str().c_str());

	//prints grain list into GID format
	for(int y = 0; y < myGrainMap.GetGrainListSize(); y++)
	{
		//prints grain ID
		myFile << y << "\t";

		//finds corresponding grain and prints euler angles
		Grain tempGrain = myGrainMap.GetGrainAtID(y);
		myFile << tempGrain.GetEulerAngle1() << "\t";
		myFile << tempGrain.GetEulerAngle2() << "\t";
		myFile << tempGrain.GetEulerAngle3() << "\t";

		//terminates line
		myFile << "\r\n";

	}
	myFile.close();
}

void FileHandler::PrintMapToKAMFile(GrainMap myGrainMap, int addendum)
{
	//sets the file name to be printed to
	ostringstream myFileName;
	myFileName << myOutputFileBase << addendum << ".kam";

	//creates the output file
	ofstream myFile(myFileName.str().c_str());

	//prints file header
	myFile << "X\tY\tLocalMisorientation\r\n";

	//prints grain map into MAP format
	for(int y = 0; y < myGrainMap.GetYSize(); y++)
	{
		for(int x = 0; x < myGrainMap.GetXSize(); x++)
		{
			//prints X & Y positions with original step adjustment, then the local KAM value
			myFile << myXStep*x << "\t";
			myFile << myYStep*y << "\t";
			myFile << myGrainMap.GetKAMAtGrid(x,y);

			//terminates line
```

```
                myFile << "\r\n";
            }
        }
        myFile.close();
}
```

```
/* MonteCarloEngine.h */
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef MONTECARLOENGINE_H_
#define MONTECARLOENGINE_H_
#include "GrainMap.h"
#include "BoundaryProperties.h"
#include "MisorientationAngles.h"


class MonteCarloEngine
{
	private:
		float systemTemp;
		vector<vector <int> >  myCoordinatePairs;
		GrainMap currentMap;
		MisorientationAngles myAngles;
		BoundaryProperties boundaryProperties;

	public:
		MonteCarloEngine(GrainMap myMap, float myTemp, int mySymmetry, int myGroup, float mobilityModifier);
		float FindBoundaryEnergy(int myID, int myX, int myY);
		GrainMap CurrentMap();
		void resetMapTo(GrainMap resetMap);

		//Time step commands

			//ignores all recrystallization behavior
			void OneTimeStepStandard();

			//RX grains expand 100% of time into non-recrystallized regions, standard otherwise (RX mobility
isotropic m = 1)
			void OneTimeStepRXOnlyHighMobility();

			//RX grains expand based on energy, released KAM energy set by factor (RX mobility isotropic m =
1)
			void OneTimeStepRXOnlyWithKAMFactorHighMobility(float myKAMfactor);

			//RX grains expand based on energy, uses a more detailed RX mobility approach with another added
scaling factor
			void OneTimeStepRXOnlyWithKAMFactorDetailedMobility(float myKAMfactor, float myMobilityFactor);

			//Same as previous but with a recovery mechanism added
			void OneTimeStepRXOnlyWithKAMFactorHighMobilityWithRecovery(float myKAMfactor, float
fractionAfterRecovery);
```

```
                void OneTimeStepRXOnlyWithKAMFactorDetailedMobilityWithRecovery(float myKAMfactor, float
myMobilityFactor,float fractionAfterRecovery);


          //Nucleation commands

                //nucleates only the high angle grain boundaries, with option to exclude twins
                void NucleateRXbyBoundary(float probability, float minAngle);
                void NucleateRXbyBoundary(float probability, float minAngle, float twinAngle, float twinWindow);

                //nucleates based upon KAM, either as a threshold or using a function
                void NucleateRXbyKAMThreshold(float recrystThreshold,float probability);
                void NucleateRXbyKAMFunction(float myKAMThreshold, float myKAMExponent, float probability);

                //permutations combining both the boundary and KAM nucleation conditions
                void NucleateRXbyBoundaryKAMThreshold(float recrystThreshold,float probability, float minAngle);
                void NucleateRXbyBoundaryKAMThreshold(float recrystThreshold,float probability, float minAngle,
float twinAngle, float twinWindow);
                void NucleateRXbyBoundaryKAMFunction(float myKAMThreshold, float myKAMExponent, float
probability, float minAngle);
                void NucleateRXbyBoundaryKAMFunction(float myKAMThreshold, float myKAMExponent, float
probability, float minAngle, float twinAngle, float twinWindow);

                //nucleates more sites based on shuffling, not for common usage
                void RandomNonRXShuffle(int everyNumPoint);
};

#endif /* MONTECARLOENGINE_H_ */
```

```
/* MonteCarloEngine.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "MonteCarloEngine.h"
#include "GrainMap.h"
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include "Neighbors.h"
#include "BoundaryProperties.h"
#include "MisorientationAngles.h"
#include <math.h>
using namespace std;

//initializes the MC Engine
MonteCarloEngine::MonteCarloEngine(GrainMap myMap, float myTemp, int mySymmetry, int myGroup, float
mobilityModifier)
{
	currentMap = myMap;
	systemTemp = myTemp;
	myAngles = MisorientationAngles(myMap, mySymmetry, myGroup);
	boundaryProperties = BoundaryProperties(mobilityModifier);

	//creates a vector of all possible coordinate pairs to be shuffled and iterated through
	vector<vector <int> >  tempCoordinatePairs;

	for(int x = 0; x < currentMap.GetXSize(); x)
	{
		for(int y = 0; y < currentMap.GetYSize(); y)
		{
			vector <int> tempCoord(2);
			tempCoord[0] = x;
			tempCoord[1] = y;
			tempCoordinatePairs.push_back(tempCoord);
		}
	}

	myCoordinatePairs = tempCoordinatePairs;

	//seeds random numbers necessary for Monte Carlo simulation
	srand (unsigned (time(0)));
}

//finds the boundary energy associate with grain with myID being located at location X,Y
```

```
float MonteCarloEngine::FindBoundaryEnergy(int myID, int myX, int myY)
{
	float myTotalEnergy = 0;

	//sums the boundary energy with each of the eight neighbors
	for(int neighbor = 0; neighbor < 8; neighbor)
	{
		//gets the ID of the grain at neighboring position
		int myNeighboringX = Neighbors::GetNeighborX(neighbor,myX);
		int myNeighboringY = Neighbors::GetNeighborY(neighbor,myY);
		int myNeighboringID = currentMap.GetIDAtGrid(myNeighboringX,myNeighboringY);

		//add the energy to the sum
		myTotalEnergy = boundaryProperties.GetEnergy(myAngles.GetAngleAt(myID,myNeighboringID));
	}

	return myTotalEnergy;
}

//returns the current state of the grain map
GrainMap MonteCarloEngine::CurrentMap()
{
	return currentMap;
}

//Must be careful when invoking to pass a map generated from the CurrentMap function from the same data set
//By storing an early simulation state with CurrentMap this can be used to run multiple simulations without
//needing to reperform the intensive initialization steps
void MonteCarloEngine::resetMapTo(GrainMap resetMap)
{
	currentMap = resetMap;
}

//runs one Monte Carlo trial on all grid locations in a random order
void MonteCarloEngine::OneTimeStepStandard()
{
	//randomly shuffles the order the grid will be tested
	random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

	//iterates through all locations on the grid
	for(size_t location = 0; location < myCoordinatePairs.size(); location)
	{
		//picks a random neighbor location
		int randomNeighbor = rand()%8;
```

```
            //finds the coordinates of the current location and the randomly picked neighbor
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
            int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);

            //finds the grain IDs at the location and selected neighbor
            int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
            int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

            //only progresses further if the two locations belong to different grains
            if(myCoordID != myNeighborID)
            {
                  //establishes necessary quantities
                  float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY);
                  float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
                  float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));

                  //probabilities used for Monte Carlo step
                  float movementProbability;
                  float probabilityThreshold = (float)rand() / RAND_MAX;

                  //if the new configuration is lower energy probability is just the mobility
                  if(alternateEnergy < currentEnergy)
                  {
                        movementProbability = boundaryMobility;
                  }
                  //or if the new energy is higher it follow an arrhenius relation
                  else
                  {
                        movementProbability = boundaryMobility*exp((currentEnergy -alternateEnergy)/systemTemp);
                  }

                  //if the attempt clears the necessary probability the location flips to the new ID
                  if(probabilityThreshold < movementProbability)
                  {
                        currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                  }
            }
      }
}

//runs one Monte Carlo trial in a random order allowing only recrystallized grains to expand, RX grains always
expand into non-recrystallized
```

```
//behaves just like standard time step if no KAM file is given (all points "recrystallized" with KAM = 0)
void MonteCarloEngine::OneTimeStepRXOnlyHighMobility()
{
	//randomly shuffles the order the grid will be tested
	random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

	//iterates through all locations on the grid
	for(size_t location = 0; location < myCoordinatePairs.size(); location)
	{
		//picks a random neighbor location
		int randomNeighbor = rand()%8;

		//finds the coordinates of the current location and the randomly picked neighbor
		int myCoordX = myCoordinatePairs[location][0];
		int myCoordY = myCoordinatePairs[location][1];
		int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
		int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);

		//finds the grain IDs at the location and selected neighbor
		int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
		int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

		//finds the KAM at the location and selected neighbor
		float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
		float neighborKAM = currentMap.GetKAMAtGrid(myNeighborX,myNeighborY);

		//only progresses further if the neighbor selected belongs to a recrystallized grain
		if(neighborKAM == 0)
		{
			//if selected location is also recrystallized proceeds as normal
			if(pickedKAM == 0)
			{
				//establishes necessary quantities
				float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY);
				float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
				float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));

				//probabilities used for Monte Carlo step
				float movementProbability;
				float probabilityThreshold = (float)rand() / RAND_MAX;

				//if the new configuration is lower energy probability is just the mobility
				if(alternateEnergy < currentEnergy)
				{
```

```
                              movementProbability = boundaryMobility;
                        }
                        //or if the new energy is higher it follow an Arrhenius relation
                        else
                        {
                              movementProbability = boundaryMobility*exp((currentEnergy -
alternateEnergy)/systemTemp);
                        }

                        //if the attempt clears the necessary probability the location flips to the new ID
                        if(probabilityThreshold < movementProbability)
                        {
                              currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                        }
                  }

                  //if selected pixel is not recrystallized, recrystallized neighbor grows into it
                  else
                  {
                        currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                  }
            }
      }
}

//runs one Monte Carlo trial in a random order allowing only recrystallized grains to expand based on KAM
(~dislocation energy) released and Read-Schockley
//mobility for recrystallization always m = 1
void MonteCarloEngine::OneTimeStepRXOnlyWithKAMFactorHighMobility(float myKAMfactor)
{
      //randomly shuffles the order the grid will be tested
      random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //picks a random neighbor location
            int randomNeighbor = rand()%8;

            //finds the coordinates of the current location and the randomly picked neighbor
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
            int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);
```

```
		//finds the grain IDs at the location and selected neighbor
		int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
		int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

		//finds the KAM at the location and selected neighbor
		float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
		float neighborKAM = currentMap.GetKAMAtGrid(myNeighborX,myNeighborY);

		//only progresses further if the neighbor selected belongs to a recrystallized grain
		if(neighborKAM == 0)
		{
			//establishes difference in energy between current and tested conditions
			//IMPORTANT : KAM energy contributes to the released energy, scaled by KAMfactor
			float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY) pickedKAM*myKAMfactor;
			float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
			float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));

			//removes the mobility barrier if the picked grain is not recrystallized, growth still limited by
the
			//probability term if KAM is relatively low.
			if(pickedKAM != 0)
			{
				boundaryMobility = 1;
			}

			//probabilities used for Monte Carlo step
			float movementProbability;
			float probabilityThreshold = (float)rand() / RAND_MAX;

			//if the new configuration is lower energy probability is just the mobility
			if(alternateEnergy < currentEnergy)
			{
				movementProbability = boundaryMobility;
			}
			//or if the new energy is higher it follows an Arrhenius relation
			else
			{
				movementProbability = boundaryMobility*exp((currentEnergy -alternateEnergy)/systemTemp);
			}

			//if the attempt clears the necessary probability the location flips to the new ID
			if(probabilityThreshold < movementProbability)
			{
```

```
                        currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                  }
            }
      }
}

//runs one Monte Carlo trial in a random order allowing only recrystallized grains to expand based on KAM
(~dislocation energy) released and Read-Schockley
//mobility is modified by a multiplicative terms of myMobilityFactor*pickedKAM following the suggestion of paper:
// Y.B. Chun, S.L Semiatin, S.K. Hwang, Acta Materialia 54 (2006) p.3673-3689
void MonteCarloEngine::OneTimeStepRXOnlyWithKAMFactorDetailedMobility(float myKAMfactor, float myMobilityFactor)
{
      //randomly shuffles the order the grid will be tested
      random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //picks a random neighbor location
            int randomNeighbor = rand()%8;

            //finds the coordinates of the current location and the randomly picked neighbor
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
            int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);

            //finds the grain IDs at the location and selected neighbor
            int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
            int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

            //finds the KAM at the location and selected neighbor
            float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
            float neighborKAM = currentMap.GetKAMAtGrid(myNeighborX,myNeighborY);

            //only progresses further if the neighbor selected belongs to a recrystallized grain
            if(neighborKAM == 0)
            {
                  //establishes difference in energy between current and tested conditions
                  //IMPORTANT : KAM energy contributes to the released energy, scaled by KAMfactor
                  float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY) pickedKAM*myKAMfactor;
                  float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
                  float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));
```

```
                  //modifies the mobility to scale with the released energy if the site is recrystallizing, like
the KAMfactor this
                  //mobility factor must be calibrated in some manner to experimental observations
                  if(pickedKAM != 0)
                  {
                        boundaryMobility = boundaryMobility*pickedKAM*myMobilityFactor;
                  }

                  //probabilities used for Monte Carlo step
                  float movementProbability;
                  float probabilityThreshold = (float)rand() / RAND_MAX;

                  //if the new configuration is lower energy probability is just the mobility
                  if(alternateEnergy < currentEnergy)
                  {
                        movementProbability = boundaryMobility;
                  }
                  //or if the new energy is higher it follows an Arrhenius relation
                  else
                  {
                        movementProbability = boundaryMobility*exp((currentEnergy -alternateEnergy)/systemTemp);
                  }

                  //if the attempt clears the necessary probability the location flips to the new ID
                  if(probabilityThreshold < movementProbability)
                  {
                        currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                  }
            }
      }
}

//runs one Monte Carlo trial in a random order allowing only recrystallized grains to expand based on KAM
(~dislocation energy) released and Read-Schockley
//mobility for recrystallization always m = 1
//recovery mechanism added
void MonteCarloEngine::OneTimeStepRXOnlyWithKAMFactorHighMobilityWithRecovery(float myKAMfactor, float
fractionAfterRecovery)
{
      //randomly shuffles the order the grid will be tested
      random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

      //iterates through all locations on the grid
```

```
	for(size_t location = 0; location < myCoordinatePairs.size(); location)
	{
		//picks a random neighbor location
		int randomNeighbor = rand()%8;

		//finds the coordinates of the current location and the randomly picked neighbor
		int myCoordX = myCoordinatePairs[location][0];
		int myCoordY = myCoordinatePairs[location][1];
		int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
		int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);

		//finds the grain IDs at the location and selected neighbor
		int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
		int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

		//finds the KAM at the location and selected neighbor
		float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
		float neighborKAM = currentMap.GetKAMAtGrid(myNeighborX,myNeighborY);

		//only progresses further if the neighbor selected belongs to a recrystallized grain
		if(neighborKAM == 0)
		{
			//establishes difference in energy between current and tested conditions
			//IMPORTANT : KAM energy contributes to the released energy, scaled by KAMfactor
			float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY) pickedKAM*myKAMfactor;
			float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
			float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));

			//removes the mobility barrier if the picked grain is not recrystallized, growth still limited by
the
			//probability term if KAM is relatively low.
			if(pickedKAM != 0)
			{
				boundaryMobility = 1;
			}

			//probabilities used for Monte Carlo step
			float movementProbability;
			float probabilityThreshold = (float)rand() / RAND_MAX;

			//if the new configuration is lower energy probability is just the mobility
			if(alternateEnergy < currentEnergy)
			{
				movementProbability = boundaryMobility;
```

```
                }
                //or if the new energy is higher it follows an Arrhenius relation
                else
                {
                      movementProbability = boundaryMobility*exp((currentEnergy -alternateEnergy)/systemTemp);
                }

                //if the attempt clears the necessary probability the location flips to the new ID
                if(probabilityThreshold < movementProbability)
                {
                      currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                      currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                }
          }

          //if the event does not represent a recrystallization attempt, recovery is undergone at the current
site which is reassigned
          //some fraction of previous value consistent with expected exponential decrease of stored energy with
time
          else
          {
                currentMap.SetKAMAtGrid(myCoordX,myCoordY,pickedKAM*fractionAfterRecovery);
          }
     }
}

//runs one Monte Carlo trial in a random order allowing only recrystallized grains to expand based on KAM
(~dislocation energy) released and Read-Schockley
//mobility is modified by a multiplicative terms of myMobilityFactor*pickedKAM following the suggestion of paper:
// Y.B. Chun, S.L Semiatin, S.K. Hwang, Acta Materialia 54 (2006) p.3673-3689
//recovery mechanism added
void MonteCarloEngine::OneTimeStepRXOnlyWithKAMFactorDetailedMobilityWithRecovery(float myKAMfactor, float
myMobilityFactor, float fractionAfterRecovery)
{
     //randomly shuffles the order the grid will be tested
     random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

     //iterates through all locations on the grid
     for(size_t location = 0; location < myCoordinatePairs.size(); location)
     {
          //picks a random neighbor location
          int randomNeighbor = rand()%8;

          //finds the coordinates of the current location and the randomly picked neighbor
          int myCoordX = myCoordinatePairs[location][0];
```

```
		int myCoordY = myCoordinatePairs[location][1];
		int myNeighborX = Neighbors::GetNeighborX(randomNeighbor,myCoordX);
		int myNeighborY = Neighbors::GetNeighborY(randomNeighbor,myCoordY);

		//finds the grain IDs at the location and selected neighbor
		int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);
		int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

		//finds the KAM at the location and selected neighbor
		float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
		float neighborKAM = currentMap.GetKAMAtGrid(myNeighborX,myNeighborY);

		//only progresses further if the neighbor selected belongs to a recrystallized grain
		if(neighborKAM == 0)
		{
			//establishes difference in energy between current and tested conditions
			//IMPORTANT : KAM energy contributes to the released energy, scaled by KAMfactor
			float currentEnergy = FindBoundaryEnergy(myCoordID, myCoordX,myCoordY) pickedKAM*myKAMfactor;
			float alternateEnergy = FindBoundaryEnergy(myNeighborID, myCoordX,myCoordY);
			float boundaryMobility =
boundaryProperties.GetMobility(myAngles.GetAngleAt(myCoordID,myNeighborID));

			//modifies the mobility to scale with the released energy if the site is recrystallizing, like
the KAMfactor this
			//mobility factor must be calibrated in some manner to experimental observations
			if(pickedKAM != 0)
			{
				boundaryMobility = boundaryMobility*pickedKAM*myMobilityFactor;
			}

			//probabilities used for Monte Carlo step
			float movementProbability;
			float probabilityThreshold = (float)rand() / RAND_MAX;

			//if the new configuration is lower energy probability is just the mobility
			if(alternateEnergy < currentEnergy)
			{
				movementProbability = boundaryMobility;
			}
			//or if the new energy is higher it follows an Arrhenius relation
			else
			{
				movementProbability = boundaryMobility*exp((currentEnergy -alternateEnergy)/systemTemp);
			}
```

```
                //if the attempt clears the necessary probability the location flips to the new ID
                if(probabilityThreshold < movementProbability)
                {
                      currentMap.SetIDAtGrid(myCoordX,myCoordY,myNeighborID);
                      currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                }
          }

          //if the event does not represent a recrystallization attempt, recovery is undergone at the current
site which is reassigned
          //some fraction of previous value consistent with expected exponential decrease of stored energy with
time
          else
          {
                currentMap.SetKAMAtGrid(myCoordX,myCoordY,pickedKAM*fractionAfterRecovery);
          }
     }
}

//sets all positions along high angle grain boundaries (above minumum angle) to recrystallized with a certain
probability
void MonteCarloEngine::NucleateRXbyBoundary(float probability, float minAngle)
{

     //iterates through all locations on the grid
     for(size_t location = 0; location < myCoordinatePairs.size(); location)
     {
          //creates random float 0 to 1
          float probabilityRX = (float)rand() / RAND_MAX;

          //finds the coordinates of the current location
          int myCoordX = myCoordinatePairs[location][0];
          int myCoordY = myCoordinatePairs[location][1];
          int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

          //tracker for number of high angle neighbors
          int highAngleNeighbors = 0;

          //iterates through all neighbors, finding the number sharing high angle boundaries
          for(int neighbor = 0 ; neighbor < 8 ; neighbor)
          {
                int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
                int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
                int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);
```

```
                //finds angle between neighbors
                float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

                //tests for a high angle grain boundary
                if(boundaryAngle < minAngle)
                {
                      //if the boundary is low angle or a twin function does nothing
                }
                else
                {
                      //keeps track if a high angle boundary is detected
                      highAngleNeighbors;
                }
          }

          //only proceeds if more than one but less than seven neighbors are high angle boundaries
          if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
          {
                //...sets location as recrystallized (KAM=0) with given probability
                if(probabilityRX < probability)
                {
                      currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                }
          }
      }
}

//sets all positions along high angle grain boundaries (above minumum angle) to recrystallize with a certain
probability
//additional functionality for a twin misorientation angle and exclusion  range
void MonteCarloEngine::NucleateRXbyBoundary(float probability, float minAngle, float twinAngle, float twinWindow)
{

      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //creates random float 0 to 1
            float probabilityRX = (float)rand() / RAND_MAX;

            //finds the coordinates of the current location
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

            //tracker for number of high angle neighbors
```

```
            int highAngleNeighbors = 0;

            //iterates through all neighbors, finding the number sharing high angle boundaries
            for(int neighbor = 0 ; neighbor < 8 ; neighbor)
            {
                  int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
                  int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
                  int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

                  //finds angle between neighbors
                  float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

                  //tests for a high angle grain boundary
                  if((boundaryAngle < minAngle) || ((boundaryAngle > twinAngle - twinWindow) && (boundaryAngle <
twinAngle  twinWindow)))
                  {
                        //if the boundary is low angle or a twin function does nothing
                  }
                  else
                  {
                        //keeps track if a high angle boundary is detected
                        highAngleNeighbors;
                  }
            }

            //only proceeds if more than one but less than seven neighbors are high angle boundaries
            if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
            {
                  //...sets location as recrystallized (KAM=0) with given probability
                  if(probabilityRX < probability)
                  {
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                  }
            }
      }
}

//sets all positions above a certain KAM threshold value to be recrystallized with a certain probability
void MonteCarloEngine::NucleateRXbyKAMThreshold(float recrystThreshold, float probability)
{

      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //creates random float 0 to 1
```

```
            float probabilityRX = (float)rand() / RAND_MAX;

            //finds the coordinates of the current location
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];

            //finds the KAM at the location
            float myKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

            //if the KAM at the location is above the threshold
            if(myKAM > recrystThreshold)
            {
                  //...sets location as recrystallized (KAM=0) with given probability
                  if(probabilityRX < probability)
                  {
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                  }
            }
      }
}

//nucleates grains based upon a function proposed in:
//Solas et al. Acta Mat. 49 (2001) p3791 & Seo et al. Comp. Mat. Sci. 43 (2008) p512
//first input sets minimum KAM for nucleation, second sets a rate exponent, third a scaling probability
void MonteCarloEngine::NucleateRXbyKAMFunction(float myKAMThreshold, float myKAMExponent, float probability)
{

      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //creates random float 0 to 1
            float probabilityRX = (float)rand() / RAND_MAX;

            //finds the coordinates of the current location
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];

            //finds the KAM at the location
            float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

            //calculates probability of nucleation at the site using an exponential function starting (rising
above 0) at a certain threshold
            float nucleateProbability = probability*(1-exp(-1*(pickedKAM - myKAMThreshold)/myKAMExponent));

            //nucleates a recrystallized grain at the current location if the condition is met
```

```
            if(probabilityRX < nucleateProbability)
            {
                  currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
            }
      }
}

//sets all positions along high angle grain boundaries (above minumum angle) above a certain KAM threshold value
to recrystallize with a certain probability
void MonteCarloEngine::NucleateRXbyBoundaryKAMThreshold(float recrystThreshold,float probability, float minAngle)
{
      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //creates random float 0 to 1
            float probabilityRX = (float)rand() / RAND_MAX;

            //finds the coordinates of the current location
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

            //tracker for number of high angle neighbors
            int highAngleNeighbors = 0;

            //iterates through all neighbors, finding the number sharing high angle boundaries
            for(int neighbor = 0 ; neighbor < 8 ; neighbor)
            {
                  int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
                  int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
                  int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

                  //finds angle between neighbors
                  float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

                  //tests for a high angle grain boundary
                  if(boundaryAngle < minAngle)
                  {
                        //if the boundary is low angle or a twin function does nothing
                  }
                  else
                  {
                        //keeps track if a high angle boundary is detected
                        highAngleNeighbors;
                  }
```

```
		}

		//only proceeds if more than one but less than seven neighbors are high angle boundaries
		if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
		{
			//...sets location as recrystallized (KAM=0) with given probability
			if(probabilityRX < probability)
			{
				//finds the KAM at the location
				float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

				//if the KAM at the location is above the threshold
				if(pickedKAM > recrystThreshold)
				{
					currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);

				}
			}
		}
	}
}

//sets all positions along high angle grain boundaries (above minumum angle) above a certain KAM threshold value
to recrystallize with a certain probability
//additional functionality for a twin misorientation angle and exclusion  range
void MonteCarloEngine::NucleateRXbyBoundaryKAMThreshold(float recrystThreshold,float probability, float minAngle,
float twinAngle, float twinWindow)
{
	//iterates through all locations on the grid
	for(size_t location = 0; location < myCoordinatePairs.size(); location)
	{
		//creates random float 0 to 1
		float probabilityRX = (float)rand() / RAND_MAX;

		//finds the coordinates of the current location
		int myCoordX = myCoordinatePairs[location][0];
		int myCoordY = myCoordinatePairs[location][1];
		int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

		//tracker for number of high angle neighbors
		int highAngleNeighbors = 0;

		//iterates through all neighbors, finding the number sharing high angle boundaries
		for(int neighbor = 0 ; neighbor < 8 ; neighbor)
		{
```

```
                int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
                int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
                int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

                //finds angle between neighbors
                float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

                //tests for a high angle grain boundary
                if((boundaryAngle < minAngle) || ((boundaryAngle > twinAngle - twinWindow) && (boundaryAngle <
twinAngle  twinWindow)))
                {
                      //if the boundary is low angle or a twin function does nothing
                }
                else
                {
                      //keeps track if a high angle boundary is detected
                      highAngleNeighbors;
                }
          }

          //only proceeds if more than one but less than seven neighbors are high angle boundaries
          if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
          {
                //...sets location as recrystallized (KAM=0) with given probability
                if(probabilityRX < probability)
                {
                      //finds the KAM at the location
                      float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

                      //if the KAM at the location is above the threshold
                      if(pickedKAM > recrystThreshold)
                      {
                                  currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                      }
                }
          }
     }
}

//sets all positions along high angle grain boundaries (above minumum angle) based upon a function proposed in:
//Solas et al. Acta Mat. 49 (2001) p3791 & Seo et al. Comp. Mat. Sci. 43 (2008) p512
//first input sets minimum KAM for nucleation, second sets a rate exponent, third a scaling probability
void MonteCarloEngine::NucleateRXbyBoundaryKAMFunction(float myKAMThreshold, float myKAMExponent, float
probability, float minAngle)
{
```

```
//iterates through all locations on the grid
for(size_t location = 0; location < myCoordinatePairs.size(); location)
{
	//creates random float 0 to 1
	float probabilityRX = (float)rand() / RAND_MAX;

	//finds the coordinates of the current location
	int myCoordX = myCoordinatePairs[location][0];
	int myCoordY = myCoordinatePairs[location][1];
	int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

	//tracker for number of high angle neighbors
	int highAngleNeighbors = 0;

	//iterates through all neighbors, finding the number sharing high angle boundaries
	for(int neighbor = 0 ; neighbor < 8 ; neighbor)
	{
		int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
		int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
		int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

		//finds angle between neighbors
		float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

		//tests for a high angle grain boundary
		if(boundaryAngle < minAngle)
		{
			//if the boundary is low angle or a twin function does nothing
		}
		else
		{
			//keeps track if a high angle boundary is detected
			highAngleNeighbors;
		}
	}

	//only proceeds if more than one but less than seven neighbors are high angle boundaries
	if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
	{
		//finds the KAM at the location
		float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

		float nucleateProbability = probability*(1-exp(-1*(pickedKAM - myKAMThreshold)/myKAMExponent));

		//nucleates a recrystallized grain at the current location if the condition is met
```

```
                if(probabilityRX < nucleateProbability)
                {
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                }
            }
      }
}

//sets all positions along high angle grain boundaries (above minumum angle) based upon a function proposed in:
//Solas et al. Acta Mat. 49 (2001) p3791 & Seo et al. Comp. Mat. Sci. 43 (2008) p512
//first input sets minimum KAM for nucleation, second sets a rate exponent, third a scaling probability
//additional functionality for a twin misorientation angle and exclusion range
void MonteCarloEngine::NucleateRXbyBoundaryKAMFunction(float myKAMThreshold, float myKAMExponent, float
probability, float minAngle, float twinAngle, float twinWindow)
{
      //iterates through all locations on the grid
      for(size_t location = 0; location < myCoordinatePairs.size(); location)
      {
            //creates random float 0 to 1
            float probabilityRX = (float)rand() / RAND_MAX;

            //finds the coordinates of the current location
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];
            int myCoordID = currentMap.GetIDAtGrid(myCoordX,myCoordY);

            //tracker for number of high angle neighbors
            int highAngleNeighbors = 0;

            //iterates through all neighbors, finding the number sharing high angle boundaries
            for(int neighbor = 0 ; neighbor < 8 ; neighbor)
            {
                  int myNeighborX = Neighbors::GetNeighborX(neighbor,myCoordX);
                  int myNeighborY = Neighbors::GetNeighborY(neighbor,myCoordY);
                  int myNeighborID = currentMap.GetIDAtGrid(myNeighborX,myNeighborY);

                  //finds angle between neighbors
                  float boundaryAngle = myAngles.GetAngleAt(myCoordID,myNeighborID);

                  //tests for a high angle grain boundary
                  if((boundaryAngle < minAngle) || ((boundaryAngle > twinAngle - twinWindow) && (boundaryAngle <
twinAngle  twinWindow)))
                  {
                        //if the boundary is low angle or a twin function does nothing
                  }
```

```
                else
                {
                        //keeps track if a high angle boundary is detected
                        highAngleNeighbors;
                }
        }

        //only proceeds if more than one but less than seven neighbors are high angle boundaries
        if((highAngleNeighbors > 1) && (highAngleNeighbors < 7))
        {
                //finds the KAM at the location
                float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);

                float nucleateProbability = probability*(1-exp(-1*(pickedKAM - myKAMThreshold)/myKAMExponent));

                //nucleates a recrystallized grain at the current location if the condition is met
                if(probabilityRX < nucleateProbability)
                {
                        currentMap.SetKAMAtGrid(myCoordX,myCoordY,0);
                }
        }
    }
}

//randomly shuffles and selects the every Nth point, if N-1th point is recrystallized and Nth point is not
//reassigns the (N-1)th point to the Nth, nucleating a new point
//Quick and dirty way to simulate nucleation due to 3D microstructure and eliminate/test some stereological
effects
void MonteCarloEngine::RandomNonRXShuffle(int everyNumPoint)
{
    //randomly shuffles the order the grid will be tested
    random_shuffle(myCoordinatePairs.begin(),myCoordinatePairs.end());

    //iterates through all locations on the grid
    for(size_t location = 0; location < myCoordinatePairs.size(); location)
    {
        //every nth iteration
        if(location % everyNumPoint == 1)
        {
            //finds the coordinates of the current location and the location before
            int myCoordX = myCoordinatePairs[location][0];
            int myCoordY = myCoordinatePairs[location][1];

            //finds the KAM at the location
            float pickedKAM = currentMap.GetKAMAtGrid(myCoordX,myCoordY);
```

```
                //only proceeds if the selected grain isn't recrystallized
                if(pickedKAM > 0)
                {
                      size_t tempLoc = location - 1;
                      int mySwapX = myCoordinatePairs[tempLoc][0];
                      int mySwapY = myCoordinatePairs[tempLoc][1];

                      //finds the grain IDs at the location and selected swap
                      int mySwapID = currentMap.GetIDAtGrid(mySwapX,mySwapY);

                      //finds the KAM at the location and selected swap
                      float swapKAM = currentMap.GetKAMAtGrid(mySwapX,mySwapY);

                      //only proceeds if the selected grain is recrystallized
                      if(swapKAM == 0)
                      {
                            //swaps the coordinates of the location before into the current location
                            currentMap.SetIDAtGrid(myCoordX,myCoordY,mySwapID);
                            currentMap.SetKAMAtGrid(myCoordX,myCoordY,swapKAM);
                      }
                }
          }
      }
}
```

```
/*GrainMap.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef GRAINMAP_H_
#define GRAINMAP_H_
#include <vector>
#include "Grain.h"
using namespace std;

class GrainMap
{
	private:
		vector<vector <int> > grainIDGrid;
		vector<vector <float> > localKAMGrid;
		vector<Grain> grainList;
		static int xSize;
		static int ySize;

	public:
		GrainMap();
		GrainMap(int myXCells, int myYCells, short int eulerAng1[], short int eulerAng2[], short int
eulerAng3[], float myKAMValues[]);
		int GetIDAtGrid(int myX, int myY);
		void SetIDAtGrid(int myX, int myY, int newID);
		Grain GetGrainAtGrid(int myX, int myY);
		Grain GetGrainAtID(int myID);
		static int GetXSize();
		static int GetYSize();
		int GetGrainListSize();
		float GetKAMAtGrid(int myX, int myY);
		void SetKAMAtGrid(int myX, int myY, float newValue);

};


#endif /* GRAINMAP_H_ */
```

```
/* GrainMap.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include <iostream>
#include "GrainMap.h"
#include <vector>
#include "Grain.h"
using namespace std;

int GrainMap::xSize;
int GrainMap::ySize;

//empty constructor for initializations
GrainMap::GrainMap(){}

GrainMap::GrainMap(int myXCells, int myYCells, short int eulerAng1[], short int eulerAng2[], short int
eulerAng3[], float myKAMValues[])
{
      //stores 2D vector dimensions GrainMap and Neighbors for future use
      xSize = myXCells;
      ySize = myYCells;

      //Initializes necessary local tracking variables
      int myArrayIndex = 0;  //1D vector index for 2D matrix
      int grainIDIndex = 0;  //ID value to be assigned to next grain

      //Initializes necessary 1 & 2D vector storage
      vector<vector <int> > tempIDGrid (myXCells, vector<int>(myYCells));
      vector<vector <float> > tempKAMGrid (myXCells, vector<float>(myYCells));
      vector<Grain> tempGrainList;

      //Transforms 1D array notation from read CTF file into a vector matrix
      for(int y = 0; y < myYCells; y++)
      {
            for(int x = 0; x < myXCells; x++)
            {
                  //finds the proper array index for the x,y matrix element
                  myArrayIndex = y*myXCells + x;

                  //reads in the KAM value at each position
                  tempKAMGrid[x][y] = myKAMValues[myArrayIndex];

                  //if the grain list is not yet populated adds the grain, sets first matrix value as well
                  if(tempGrainList.empty())
                  {
```

```
	tempGrainList.push_back(Grain(grainIDIndex,eulerAng1[myArrayIndex],eulerAng2[myArrayIndex],eulerAng3[myArray
Index]));
				tempIDGrid[x][y] = grainIDIndex;
				grainIDIndex++;
			}
			else
			{
				//creates a temporary trigger set to what the next grain ID would be (i.e. higher than any
existing ID)
				int tempGrainIDMatch = grainIDIndex;

				//goes through all known grains at time of execution
				for(size_t i = 0; i < tempGrainList.size(); i++)
				{
					Grain tempGrain = tempGrainList[i];

					//checks to see if any of the grains have identical euler angles to the grain at the
list position
					if(eulerAng1[myArrayIndex] == tempGrain.GetEulerAngle1())
					{
						if(eulerAng2[myArrayIndex] == tempGrain.GetEulerAngle2())
						{
							if(eulerAng3[myArrayIndex] == tempGrain.GetEulerAngle3())
							{
								//if a grain does match, keeps the ID
								tempGrainIDMatch = tempGrain.GetID();
							}
						}
					}
				}

				// if the euler values matched a previous grain passes the proper ID into the grid
				// otherwise adds a new grain to the grain list and passes the new ID into the grid
				if(tempGrainIDMatch < grainIDIndex)
				{
					tempIDGrid[x][y] = tempGrainIDMatch;
				}
				else
				{

	tempGrainList.push_back(Grain(grainIDIndex,eulerAng1[myArrayIndex],eulerAng2[myArrayIndex],eulerAng3[myArray
Index]));
					tempIDGrid[x][y] = grainIDIndex;
					grainIDIndex++;
```

```
                        }
                    }

                    //tracker
                    cout << x << " " << y << " position mapped\n";
                }
            }

            // Pass populated list of grains and a 2D matrices mapping their ID & KAM values up to the GrainMap
            grainIDGrid = tempIDGrid;
            grainList = tempGrainList;
            localKAMGrid = tempKAMGrid;

            //Program Status Update
            cout << "Grain Map Created\n";
}

//returns the ID of location X,Y
int GrainMap::GetIDAtGrid(int myX, int myY)
{
            return grainIDGrid[myX][myY];
}

//returns the Grain at location X,Y
Grain GrainMap::GetGrainAtGrid(int myX, int myY)
{
            return grainList[grainIDGrid[myX][myY]];
}

//returns the Grain of given ID
Grain GrainMap::GetGrainAtID(int myID)
{
            return grainList[myID];
}

//sets the ID of location X,Y to a new value
void GrainMap::SetIDAtGrid(int myX, int myY, int newID)
{
            grainIDGrid[myX][myY] = newID;
}

//returns the x dimension of the grid
int GrainMap::GetXSize()
{
            return xSize;
```

```
}

//returns the y dimension of the grid
int GrainMap::GetYSize()
{
	return ySize;
}

//returns the size of the grain list
int GrainMap::GetGrainListSize()
{
	return grainList.size();
}

//returns the KAM at location X,Y
float GrainMap::GetKAMAtGrid(int myX, int myY)
{
	return localKAMGrid[myX][myY];
}

//sets the KAM of location X,Y to a new value
void GrainMap::SetKAMAtGrid(int myX, int myY, float newValue)
{
	localKAMGrid[myX][myY] = newValue;
}
```

```
/* Grain.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef GRAIN_H_
#define GRAIN_H_
#include <vector>
#include "Matrix.h"

class Grain
{
	private:
		int myID;
		short int eulerAngle1;
		short int eulerAngle2;
		short int eulerAngle3;
		Matrix orientationMatrix;

	public:
		Grain(int grainID, short int eulerAng1, short int eulerAng2, short int eulerAng3);
		int GetID();
		short int GetEulerAngle1();
		short int GetEulerAngle2();
		short int GetEulerAngle3();
		Matrix GetOrientationMatrix();
};

#endif /* GRAIN_H_ */
```

```
/* Grain.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "Grain.h"
#include <algorithm>
#include "Matrix.h"
#include <iostream>
using namespace std;

Grain::Grain(int grainID, short int eulerAng1, short int eulerAng2, short int eulerAng3)
{
      myID = grainID;
      eulerAngle1 = eulerAng1;
      eulerAngle2 = eulerAng2;
      eulerAngle3 = eulerAng3;
      orientationMatrix = Matrix(eulerAng1,eulerAng2,eulerAng3);
}

int Grain::GetID()
{
      return myID;
}

short int Grain::GetEulerAngle1()
{
      return eulerAngle1;
}

short int Grain::GetEulerAngle2()
{
      return eulerAngle2;
}

short int Grain::GetEulerAngle3()
{
      return eulerAngle3;
}

Matrix Grain::GetOrientationMatrix()
{
      return orientationMatrix;
}
```

```
/*MisorientationAngles.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef MISORIENTATIONANGLES_H_
#define MISORIENTATIONANGLES_H_
#include <vector>
#include "GrainMap.h"
#include "Matrix.h"
using namespace std;

class MisorientationAngles
{
      private:
            vector<vector <short int> > angleBetween;
            vector <Matrix> symmetryOperators;
            int grainListLength;

      public:
            MisorientationAngles();
            MisorientationAngles(GrainMap myMap, int symmetryBranch, int symmetryGroup);
            float FindAngle(Matrix firstMatrix, Matrix secondMatrix);
            float GetLowestAngle(Matrix firstMatrix, Matrix secondMatrix);
            int GetDimension();
            float GetAngleAt(int myIDX, int myIDY);
};

#endif /* MISORIENTATIONANGLES_H_ */
```

```
/* MisorientationAngles.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "MisorientationAngles.h"
#include "Matrix.h"
#include <math.h>
#include <vector>
#include <iostream>
using namespace std;

//empty constructor
MisorientationAngles::MisorientationAngles(){grainListLength =0;}

MisorientationAngles::MisorientationAngles(GrainMap myMap, int symmetryBranch, int symmetryGroup)
{
      vector <Matrix> myOperators;

      //Contains symmetry operators for hexagonal branch space groups
      if(symmetryBranch == 6)
      {

            //hard codes all possible operators for branch 6
            float sqrt3d2 = sqrt(3)/2;

            float tempMat1[3][3] = {{1, 0, 0},
                                    {0, 1, 0},
                                    {0, 0, 1}};

            float tempMat2[3][3] = {{-.5, sqrt3d2, 0},
                                    {-sqrt3d2, -.5, 0},
                                    {0, 0, 1}};

            float tempMat3[3][3] = {{-.5, -sqrt3d2, 0},
                                    {sqrt3d2, -.5, 0},
                                    {0, 0, 1}};

            float tempMat4[3][3] = {{.5, sqrt3d2, 0},
                                    {-sqrt3d2, .5, 0},
                                    {0, 0, 1}};

            float tempMat5[3][3] = {{-1, 0, 0},
                                    {0, -1, 0},
                                    {0, 0, 1}};

            float tempMat6[3][3] = {{.5, -sqrt3d2, 0},
```

```
                                    {sqrt3d2, .5, 0},
                                    {0, 0, 1}};

float tempMat7[3][3] = {{-.5, -sqrt3d2, 0},
                                    {-sqrt3d2, .5, 0},
                                    {0, 0, -1}};

float tempMat8[3][3] = {{1, 0, 0},
                                    {0, -1, 0},
                                    {0, 0, -1}};

float tempMat9[3][3] = {{-.5, sqrt3d2, 0},
                                    {sqrt3d2, .5, 0},
                                    {0, 0, -1}};

float tempMat10[3][3] =     {{.5, sqrt3d2, 0},
                                    {sqrt3d2, -.5, 0},
                                    {0, 0, -1}};

float tempMat11[3][3] =     {{-1, 0, 0},
                                    {0, 1, 0},
                                    {0, 0, -1}};

float tempMat12[3][3] =     {{.5, -sqrt3d2, 0},
                                    {-sqrt3d2, -.5, 0},
                                    {0, 0, -1}};


//chooses the proper operators for the given group
if(symmetryGroup == 622)
{
      myOperators.push_back(Matrix(tempMat1));
      myOperators.push_back(Matrix(tempMat2));
      myOperators.push_back(Matrix(tempMat3));
      myOperators.push_back(Matrix(tempMat4));
      myOperators.push_back(Matrix(tempMat5));
      myOperators.push_back(Matrix(tempMat6));
      myOperators.push_back(Matrix(tempMat7));
      myOperators.push_back(Matrix(tempMat8));
      myOperators.push_back(Matrix(tempMat9));
      myOperators.push_back(Matrix(tempMat10));
      myOperators.push_back(Matrix(tempMat11));
      myOperators.push_back(Matrix(tempMat12));
}
```

```
	if(symmetryGroup == 32)
	{
		myOperators.push_back(Matrix(tempMat1));
		myOperators.push_back(Matrix(tempMat2));
		myOperators.push_back(Matrix(tempMat3));
		myOperators.push_back(Matrix(tempMat10));
		myOperators.push_back(Matrix(tempMat11));
		myOperators.push_back(Matrix(tempMat12));
	}

	if(symmetryGroup == 6)
	{
		myOperators.push_back(Matrix(tempMat1));
		myOperators.push_back(Matrix(tempMat2));
		myOperators.push_back(Matrix(tempMat3));
		myOperators.push_back(Matrix(tempMat4));
		myOperators.push_back(Matrix(tempMat5));
		myOperators.push_back(Matrix(tempMat6));
	}

	if(symmetryGroup == 3)
	{
		myOperators.push_back(Matrix(tempMat1));
		myOperators.push_back(Matrix(tempMat2));
		myOperators.push_back(Matrix(tempMat3));
	}

}

//Contains symmetry operators for cubic branch space groups
if(symmetryBranch == 3)
{
	float tempMat1[3][3] = {{1, 0, 0},
				{0, 1, 0},
				{0, 0, 1}};

	float tempMat2[3][3] = {{0, 0, 1},
				{1, 0, 0},
				{0, 1, 0}};

	float tempMat3[3][3] = {{0, 1, 0},
				{0, 0, 1},
				{1, 0, 0}};

	float tempMat4[3][3] = {{0, -1, 0},
```

```
                                    {0, 0, 1},
                                    {-1, 0, 0}};

float tempMat5[3][3] = {{0, -1, 0},
                                    {0, 0, -1},
                                    {1, 0, 0}};

float tempMat6[3][3] = {{0, 1, 0},
                                    {0, 0, -1},
                                    {-1, 0, 0}};

float tempMat7[3][3] = {{0, 0, -1},
                                    {1, 0, 0},
                                    {0, -1, 0}};

float tempMat8[3][3] = {{0, 0, -1},
                                    {-1, 0, 0},
                                    {0, 1, 0}};

float tempMat9[3][3] = {{0, 0, 1},
                                    {-1, 0, 0},
                                    {0, -1, 0}};

float tempMat10[3][3] =      {{-1, 0, 0},
                                    {0, 1, 0},
                                    {0, 0, -1}};

float tempMat11[3][3] =      {{-1, 0, 0},
                                    {0, -1, 0},
                                    {0, 0, 1}};

float tempMat12[3][3] =      {{1, 0, 0},
                                    {0, -1, 0},
                                    {0, 0, -1}};

float tempMat13[3][3] =     {{0, 0, -1},
                                    {0, -1, 0},
                                    {-1, 0, 0}};

float tempMat14[3][3] =      {{0, 0, 1},
                                    {0, -1, 0},
                                    {1, 0, 0}};

float tempMat15[3][3] =      {{0, 0, 1},
                                    {0, 1, 0},
```

```
                                    {-1, 0, 0}};

float tempMat16[3][3] =      {{0, 0, -1},
                              {0, 1, 0},
                              {1, 0, 0}};

float tempMat17[3][3] =      {{-1, 0, 0},
                              {0, 0, -1},
                              {0, -1, 0}};

float tempMat18[3][3] =      {{1, 0, 0},
                              {0, 0, -1},
                              {0, 1, 0}};

float tempMat19[3][3] =      {{1, 0, 0},
                              {0, 0, 1},
                              {0, -1, 0}};

float tempMat20[3][3] =      {{-1, 0, 0},
                              {0, 0, 1},
                              {0, 1, 0}};

float tempMat21[3][3] =      {{0, -1, 0},
                              {-1, 0, 0},
                              {0, 0, -1}};

float tempMat22[3][3] =      {{0, 1, 0},
                              {-1, 0, 0},
                              {0, 0, 1}};

float tempMat23[3][3] =      {{0, 1, 0},
                              {1, 0, 0},
                              {0, 0, -1}};

float tempMat24[3][3] =      {{0, -1, 0},
                              {1, 0, 0},
                              {0, 0, 1}};

//chooses the proper operators for the given group
if(symmetryGroup == 432)
{
      myOperators.push_back(Matrix(tempMat1));
      myOperators.push_back(Matrix(tempMat2));
      myOperators.push_back(Matrix(tempMat3));
      myOperators.push_back(Matrix(tempMat4));
```

```
	myOperators.push_back(Matrix(tempMat5));
	myOperators.push_back(Matrix(tempMat6));
	myOperators.push_back(Matrix(tempMat7));
	myOperators.push_back(Matrix(tempMat8));
	myOperators.push_back(Matrix(tempMat9));
	myOperators.push_back(Matrix(tempMat10));
	myOperators.push_back(Matrix(tempMat11));
	myOperators.push_back(Matrix(tempMat12));
	myOperators.push_back(Matrix(tempMat13));
	myOperators.push_back(Matrix(tempMat14));
	myOperators.push_back(Matrix(tempMat15));
	myOperators.push_back(Matrix(tempMat16));
	myOperators.push_back(Matrix(tempMat17));
	myOperators.push_back(Matrix(tempMat18));
	myOperators.push_back(Matrix(tempMat19));
	myOperators.push_back(Matrix(tempMat20));
	myOperators.push_back(Matrix(tempMat21));
	myOperators.push_back(Matrix(tempMat22));
	myOperators.push_back(Matrix(tempMat23));
	myOperators.push_back(Matrix(tempMat24));
}

if(symmetryGroup == 23)
{
	myOperators.push_back(Matrix(tempMat1));
	myOperators.push_back(Matrix(tempMat2));
	myOperators.push_back(Matrix(tempMat3));
	myOperators.push_back(Matrix(tempMat4));
	myOperators.push_back(Matrix(tempMat5));
	myOperators.push_back(Matrix(tempMat6));
	myOperators.push_back(Matrix(tempMat7));
	myOperators.push_back(Matrix(tempMat8));
	myOperators.push_back(Matrix(tempMat9));
	myOperators.push_back(Matrix(tempMat10));
	myOperators.push_back(Matrix(tempMat11));
	myOperators.push_back(Matrix(tempMat12));
}

if(symmetryGroup == 3)
{
	myOperators.push_back(Matrix(tempMat1));
	myOperators.push_back(Matrix(tempMat2));
	myOperators.push_back(Matrix(tempMat3));
	myOperators.push_back(Matrix(tempMat4));
	myOperators.push_back(Matrix(tempMat5));
```

```
            myOperators.push_back(Matrix(tempMat6));
            myOperators.push_back(Matrix(tempMat7));
            myOperators.push_back(Matrix(tempMat8));
            myOperators.push_back(Matrix(tempMat9));
      }
}

//Contains symmetry operators for tetragonal and lower space group branches
if(symmetryBranch == 4)
{
      float tempMat1[3][3] = {{1, 0, 0},
                              {0, 1, 0},
                              {0, 0, 1}};

      float tempMat2[3][3] = {{-1, 0, 0},
                              {0, 1, 0},
                              {0, 0, -1}};

      float tempMat3[3][3] = {{1, 0, 0},
                              {0, -1, 0},
                              {0, 0, -1}};

      float tempMat4[3][3] = {{-1, 0, 0},
                              {0, -1, 0},
                              {0, 0, 1}};

      float tempMat5[3][3] = {{0, 1, 0},
                              {-1, 0, 0},
                              {0, 0, 1}};

      float tempMat6[3][3] = {{0, -1, 0},
                              {1, 0, 0},
                              {0, 0, 1}};

      float tempMat7[3][3] = {{0, 1, 0},
                              {1, 0, 0},
                              {0, 0, -1}};

      float tempMat8[3][3] = {{0, -1, 0},
                              {-1, 0, 0},
                              {0, 0, -1}};

      //chooses the proper operators for the given group
      if(symmetryGroup == 42)
      {
```

```
            myOperators.push_back(Matrix(tempMat1));
            myOperators.push_back(Matrix(tempMat2));
            myOperators.push_back(Matrix(tempMat3));
            myOperators.push_back(Matrix(tempMat4));
            myOperators.push_back(Matrix(tempMat5));
            myOperators.push_back(Matrix(tempMat6));
            myOperators.push_back(Matrix(tempMat7));
            myOperators.push_back(Matrix(tempMat8));
      }

      if(symmetryGroup == 4)
      {
            myOperators.push_back(Matrix(tempMat1));
            myOperators.push_back(Matrix(tempMat4));
            myOperators.push_back(Matrix(tempMat5));
            myOperators.push_back(Matrix(tempMat6));
      }

      //aka orthogonal
      if(symmetryGroup == 222)
      {
            myOperators.push_back(Matrix(tempMat1));
            myOperators.push_back(Matrix(tempMat2));
            myOperators.push_back(Matrix(tempMat3));
            myOperators.push_back(Matrix(tempMat4));
      }

      //aka Monoclinic
      if(symmetryGroup == 2)
      {
            myOperators.push_back(Matrix(tempMat1));
            myOperators.push_back(Matrix(tempMat2));
      }

      //aka Triclinic
      if(symmetryGroup == 1)
      {
            myOperators.push_back(Matrix(tempMat1));
      }
}

//finish of symmetry determination section, stores vector
symmetryOperators = myOperators;

//stores the length of the vector dimensions
```

```
	grainListLength = myMap.GetGrainListSize();

	//initializes a temporary grid for storing angles between grains ID x and ID y
	vector<vector <short int> > tempAngleGrid (grainListLength, vector<short int>(grainListLength));

	//finds misorientation between all sets of grains
	/*
	 * While this function could be more efficient, storing only one half the matrix or being calculated when
	 * grains actually touch in the simulation, the storage is low and I'm not convinced calling an if statement
	 * every simulation step to see if the value in this array exists already is faster than doing them all
once.
	 */
	for(int x = 0; x < grainListLength; x++)
	{
		//sets diagonal component to zero
		tempAngleGrid[x][x] = 0;

		cout << x << " grainID\n";

		//sets x,y and y,x locations to the misorientation angle between the x and y grains
		for(int y = x+1; y < grainListLength; y++)
		{
			Matrix tempMatX = myMap.GetGrainAtID(x).GetOrientationMatrix();
			Matrix tempMatY = myMap.GetGrainAtID(y).GetOrientationMatrix();

			float misorientAngle = GetLowestAngle(tempMatX,tempMatY);

			//rounds to nearest int
			misorientAngle = roundf(misorientAngle);

			tempAngleGrid[x][y] = misorientAngle;
			tempAngleGrid[y][x] = misorientAngle;
		}
	}

	//stores the angle matrix
	angleBetween = tempAngleGrid;

	//Program Status Update
	cout << "Misorientation Angles Calculated\n";
}

//finds the angle in degrees between two matrices
float MisorientationAngles::FindAngle(Matrix firstMatrix, Matrix secondMatrix)
{
```

```
	Matrix misorientMat = firstMatrix.Multiply(secondMatrix.Transpose());

	return 180 * acos( (misorientMat.GetValue(0,0) + misorientMat.GetValue(1,1) + misorientMat.GetValue(2,2)-
1)/2 ) / 3.1416 ;
}

//finds the lowest misorientation angle in degrees between two matrices utilizing symmetry
float MisorientationAngles::GetLowestAngle(Matrix firstMatrix, Matrix secondMatrix)
{
	float tempLowestAngle = 360;

	//tests all possibly symmetry conditions to find the lowest misorientation angle
	for(size_t symOp = 0; symOp < symmetryOperators.size(); symOp ++)
	{
		float tempCurrentAngle = FindAngle(firstMatrix, symmetryOperators[symOp].Multiply(secondMatrix));

		if(tempCurrentAngle < tempLowestAngle)
		{
			tempLowestAngle = tempCurrentAngle;
		}
	}

	return tempLowestAngle;
}

//returns the dimension of the grain list
int MisorientationAngles::GetDimension()
{
	return grainListLength;
}

//returns the dimension of the grain list
float MisorientationAngles::GetAngleAt(int myIDX, int myIDY)
{
	return angleBetween[myIDX][myIDY];
}
```

```
/* BoundaryProperties.h */
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef BOUNDARYPROPERTIES_H_
#define BOUNDARYPROPERTIES_H_
#include <vector>
#include "MisorientationAngles.h"

class BoundaryProperties
{
      private:
            vector<float> boundaryEnergy;
            vector<float> boundaryMobility;

      public:
            BoundaryProperties();
            BoundaryProperties(float mobilityModifier);
            float FindEnergy(int myAngle);
            float FindMobility(int myAngle, float mobilityModifier);
            float GetEnergy(int myAngle);
            float GetMobility(int myAngle);
};

#endif  /* BOUNDARYPROPERTIES_H_ */
```

```
/*BoundaryProperties.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "BoundaryProperties.h"
#include <math.h>
#include <iostream>

//empty constructor, for initialization use in temporary code loops
BoundaryProperties::BoundaryProperties(){}

//initialization, different implementation that the first version of the MC program
//now rounds all misorientation angles to the nearest degree for large computation/memory savings
BoundaryProperties::BoundaryProperties(float mobilityModifier)
{
	//initializes temporary grids for storing the calculated energies and mobilities
	//currently for int angles 0-360, can probably be reduced to angles 0-180 instead
	vector<float> tempEnergyGrid (361);
	vector<float> tempMobilityGrid (361);

	//iterates through the vectors calculating values
	for(int x = 0; x < 361; x++)
	{
		//calculates the boundary energy and mobility for misorientation angles 0-360
		tempEnergyGrid[x] = FindEnergy(x);
		tempMobilityGrid[x] = FindMobility(x,mobilityModifier);
	}

	//stores values
	boundaryEnergy = tempEnergyGrid;
	boundaryMobility = tempMobilityGrid;

	//Program Status Update
	cout << "Energies and Mobilities Calculated\n";
}


//finds the relative grain boundary energy
float BoundaryProperties::FindEnergy(int myAngle)
{
	//according to the Read-Shockley equation

	float temp = myAngle;

	if (temp < 15)
	{
```

```
            if(temp == 0)
            {
                  return 0;
            }
            else
            {
                  return (temp/15)*(1-log(temp/15));
            }
      }
      else
      {
            return 1;
      }
}

//find the relative mobility of the grain boundary, empirical model for general boundaries
float BoundaryProperties::FindMobility(int myAngle, float mobilityModifier)
{
      //modifier used to slow grain growth if necessary
      float mobilityAdjustment = mobilityModifier;

      float temp = myAngle;

      if (temp < 15)
      {
            if(temp == 0)
            {
                  return 0;
            }
            else
            {
                  return (mobilityAdjustment*(1-exp(-5*pow((temp/15),4)))/(1-exp(-5)));
            }
      }
            else
      {
            return mobilityAdjustment;
      }
}

//returns the boundary energy value for a given misorientation angle
float BoundaryProperties::GetEnergy(int myAngle)
{
      return boundaryEnergy[myAngle];
}
```

```
//returns the boundary mobility value for a given misorientation angle
float BoundaryProperties::GetMobility(int myAngle)
{
      return boundaryMobility[myAngle];
}
```

```
/*Neighbors.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef NEIGHBORS_H_
#define NEIGHBORS_H_

class Neighbors
{
      public:
            static int GetNeighborX(int myPosition, int myX);
            static int GetNeighborY(int myPosition, int myY);
};

#endif /* NEIGHBORS_H_ */
```

```
/* Neighbors.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "Neighbors.h"
#include "GrainMap.h"

// Finds neighboring grid spaces while imposing periodic boundary conditions

//  Position map of neighbors :
//  0  1  2
//  3  X  4
//  5  6  7

int Neighbors::GetNeighborX(int myPosition, int myX)
{
      int tempGridXSize = GrainMap::GetXSize();

      if(myPosition == 2 || myPosition == 4 || myPosition == 7)
      {
            return (myX+1)%tempGridXSize;
      }
      else if(myPosition == 1 || myPosition == 6)
      {
            return myX;
      }
      else
      {
            return (myX+tempGridXSize-1)%tempGridXSize;
      }
}

int Neighbors::GetNeighborY(int myPosition, int myY)
{
      int tempGridYSize = GrainMap::GetYSize();

      if(myPosition == 5 || myPosition == 6 || myPosition == 7)
      {
            return (myY+1)%tempGridYSize;
      }
      else if(myPosition == 3 || myPosition == 4)
      {
            return myY;
      }
      else
      {
```

```
            return (myY+tempGridYSize-1)%tempGridYSize;
        }
}
```

```
/* Matrix.h*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#ifndef MATRIX_H_
#define MATRIX_H_

class Matrix
{
	private:
		float myMatrix[3][3];

	public:
		Matrix();
		Matrix(float inputMatrix[3][3]);
		Matrix(float euler1, float euler2, float euler3);
		void PrintMatrix();
		Matrix Multiply(Matrix secondMatrix);
		void SetValue(int myX, int myY, float myValue);
		float GetValue(int myX, int myY);
		float GetDeterminant();
		Matrix Transpose();
};

#endif /* MATRIX_H_ */
```

```
/* Matrix.cpp*/
//Written by Matt Steiner, Ph.D (2016). Instructions & citation info in ReadMe.txt

#include "Matrix.h"
#include <iostream>
#include <string.h>
#include <math.h>
#define PI 3.14159265
using namespace std;


//empty constructor, for initialization use in temporary code loops
Matrix::Matrix(){}

//constructor if input given as a matrix
Matrix::Matrix(float inputMatrix[3][3])
{
      //copies input into the stored matrix
      memcpy(myMatrix,inputMatrix,sizeof(myMatrix));
}

//constructor if input given as three euler angles
//IMPORTANT: CTF Files gives euler angles in Z-X-Z Bunge Convention!!
Matrix::Matrix(float euler1, float euler2, float euler3)
{
      //converts angles into radian form
      float eulerRad1 = euler1*PI/180;
      float eulerRad2 = euler2*PI/180;
      float eulerRad3 = euler3*PI/180;

      //creates a rotation matrix from reference frame using Z-X-Z convention
      myMatrix[0][0] = cos(eulerRad1)*cos(eulerRad3) - cos(eulerRad2)*sin(eulerRad1)*sin(eulerRad3);
      myMatrix[0][1] = -1*cos(eulerRad1)*sin(eulerRad3) - cos(eulerRad2)*cos(eulerRad3)*sin(eulerRad1) ;
      myMatrix[0][2] = sin(eulerRad1)*sin(eulerRad2);
      myMatrix[1][0] = cos(eulerRad3)*sin(eulerRad1) + cos(eulerRad1)*cos(eulerRad2)*sin(eulerRad3);
      myMatrix[1][1] = cos(eulerRad1)*cos(eulerRad2)*cos(eulerRad3) - sin(eulerRad1)*sin(eulerRad3);
      myMatrix[1][2] = -1*cos(eulerRad1)*sin(eulerRad2);
      myMatrix[2][0] = sin(eulerRad2)*sin(eulerRad3);
      myMatrix[2][1] = cos(eulerRad3)*sin(eulerRad2);
      myMatrix[2][2] = cos(eulerRad2);
}

//prints matrix to console for debugging purposes
void Matrix::PrintMatrix()
{
```

```
	for(int x = 0; x < 3; x++)
	{
		for(int y = 0; y < 3; y++)
		{
			cout << myMatrix[x][y] << "\t";
		}
		cout << "\n";
	}
}

//multiplies two 3x3 matrices and returns the result
Matrix Matrix::Multiply(Matrix secondMatrix)
{
	float product[3][3];

	// This code freakishly blows up for only the bottom left corner value, don't ask me why
	// something intermittent about += -0
	/*for(int row = 0; row < 3; row++)
	{
		for(int col = 0; col < 3; col++)
		{
			//multiplies row with respective column, sums into product matrix
			for(int inner = 0; inner < 3; inner++)
			{
				product[row][col]+= myMatrix[row][inner]*secondMatrix.GetValue(inner,col);
			}
		}
	}*/

	//This code fixes the above problem, don't try to simplify
	for(int row = 0; row < 3; row++)
	{
		for(int col = 0; col < 3; col++)
		{
				float test1 = myMatrix[row][0]*secondMatrix.GetValue(0,col);
				float test2 = myMatrix[row][1]*secondMatrix.GetValue(1,col);
				float test3 = myMatrix[row][2]*secondMatrix.GetValue(2,col);

				product[row][col] = test1+test2+test3;
		}
	}
	return Matrix(product);
}

//returns the determinate of the matrix
```

```
float Matrix::GetDeterminant()
{
	float tempDeterm = 0;
	tempDeterm += myMatrix[0][0]*myMatrix[1][1]*myMatrix[2][2];
	tempDeterm += myMatrix[0][1]*myMatrix[1][2]*myMatrix[2][0];
	tempDeterm += myMatrix[0][2]*myMatrix[1][0]*myMatrix[2][1];
	tempDeterm -= myMatrix[0][2]*myMatrix[1][1]*myMatrix[2][0];
	tempDeterm -= myMatrix[0][1]*myMatrix[1][0]*myMatrix[2][2];
	tempDeterm -= myMatrix[0][0]*myMatrix[1][2]*myMatrix[2][1];

	return tempDeterm;
}

//returns the transpose of the matrix
Matrix Matrix::Transpose()
{
	Matrix tempMatrix;

	for(int x = 0; x < 3 ; x++)
	{
		for(int y = 0; y < 3; y++)
		{
			tempMatrix.SetValue(x,y,myMatrix[y][x]);
		}
	}

	return tempMatrix;
}

//returns value at X,Y
float Matrix::GetValue(int myX, int myY)
{
	return myMatrix[myX][myY];
}

//sets value at X,Y
void Matrix::SetValue(int myX, int myY, float myValue)
{
	myMatrix[myX][myY] = myValue;
}
```