Interface Texture Development During Grain Growth

Jason Gruber

Department of Materials Science and Engineering

Carnegie Mellon University

Contents

Abstract

Simulations of 2D and 3D grain growth with misorientation dependent anisotropic interfacial energy and mobility were performed using standard numerical methods. Average grain size, grain size distribution, and area and number weighted misorientation distribution functions (MDFs) were computed at equal time intervals throughout each simulation. The initial microstructures for all simulations were produced through isostropic coarsening of a domain with all single pixel/voxel grains. Grain orientations were either assigned randomly or were chosen to produce a single component orientation texture. Various combinations of energy and mobility functions were used, in particular isotropic (constant value), Read-Shockley type, and step functions. The simulations were validated by MDF measurements in polycrystalline magnesia.

Simulations of 3D grain growth with inclination dependent properties were performed using the moving finite element method with a microstructure discretized as a tetrahedral mesh. In these simulations, the grain boundary character distribution (GBCD) was measured. The initial microstructure was produced through isotropic coarsening with randomly assigned subdomains. Grain orientations were assigned randomly. Energy and mobility functions used took the form of a summation over the values of a given function for the interface plane in either grain reference system. The simulations were validated by comparison with the measured GBCD in magnesia.

In simulations with misorientation dependent properties and random initial orientation texture, both the area and number weighted MDFs reached steady-state distributions after a moderate amount of grain growth. Similar qualitative results are found in all cases, regardless of the functional form of the boundary properties, crystal symmetry, or dimensionality. Grain boundaries with relatively low energy have larger average areas occur in greater number than those with higher relative energies. Mobility anisotropy has no measureable effect on interface texture in the simulations performed. The relationship between grain boundary energy and the relative average area of grain boundaries is found to be approximately one-to-one. Interface texture development occurs by changes in the relative average areas of grain boundaries, explained by a triple junction lengthening model, as well as biased elimination of grain boundaries through topological events. In systems with misorientation dependent properties and non-random intial orientation texture, the MDFs do not reach steady states. Interface texture in such cases occurs by the mechanisms suggested above but is also enhanced by strengthening orientation texture.

A quantitative critical event model that is in good agreement with all simulations is presented. This model predicts that the number weighted MDF $f_N(\theta, t)$, the area weighted MDF $f_A(\theta, t)$, the texture weighted MDF $f_0(\theta, t)$, and the relative average area of grain boundaries $\langle A\rangle(\theta, t)/\langle A\rangle$ are related by the expressions

$$f_N(\theta, t) \propto f_0(\theta, t)\langle A\rangle(\theta, t)/\langle A\rangle$$

and

$$f_A(\theta, t) \propto f_0(\theta, t)\left[\langle A\rangle(\theta, t)/\langle A\rangle\right]^2 .$$

Results from simulations with inclination dependent anisotropy suggest a similar mechanism as in the case of misorientation dependent anisotropy, with boundaries of lowest energy occuring with the highest frequencies, and no apparent effect of mobility anisotropy.

List of Symbols

Many of the following symbols that represent functions also appear in the text with explicit time dependence.

t	Time variable.
$\varphi_1, \Phi, \varphi_2$	Euler angles (Bunge convention).
Δg	Misorientation (rotation).
ϕ, θ	Spherical coordinates.
$\mathbf{n}$	Surface normal vector.
θ	Disorientation angle.
$\gamma(\theta)$	Interfacial energy as a function of disorientation angle.
$M(\theta)$	Interfacial mobility as a function of disorientation angle.
v_n	Normal velocity of a moving interface.
ΔE	Energy change with respect to interface motion.
kT	Thermal energy (Boltzmann constant times temperature).
β	Thermal scaling constant used in Monte Carlo simulations.
$\chi(i,j)$	Near neighbor indication function used in Monte Carlo simulations.
$\epsilon^{\text{tri}}, \epsilon^{\text{tet}}$	Scaling constants used in moving finite element simulations.
$Q^{\text{tri}}, Q^{\text{tet}}$	Element quality energies used in moving finite element simulations.
A	Area of triangle element, grain boundary area.
V	Volume of tetrahedron element, grain volume.

N	The total number of interfaces in a system.
$N(\theta)$	The total number of interfaces characterized by disorientation angle θ.
$\langle A \rangle$	The average area (length) of all interfaces in a system.
$\langle A \rangle(\theta)$	The average area (length) of interfaces characterized by disorientation angle θ.
$f_N(\theta)$	Number weighted misorientation distribution function as a probability density.
$\lambda_N(\theta)$	Number weighted misorientation distribution function in multiples random.
$f_A(\theta)$	Area weighted misorientation distribution function as a probability density.
$\lambda_A(\theta)$	Area weighted misorientation distribution function in multiples random.
$f_\circ(\theta)$	Texture weighted misorientation distribution function.

List of Figures

Chapter 1

Introduction

1.1 Motivation

The motivation for studying materials processing is a desire to predict and control the structure, and consequently, the utility of engineered products. The majority of engineering materials are polycrystalline, and for many such materials the microstructure largely determines the macroscopic physical properties. A systematic study of materials processing and resulting microstructures then has obvious value, and makes up a large part of the scientific literature. In this work we examine grain growth, a process common to all polycrystalline materials, and its effect on several quantitative microstructural measures.

Recent measurements of microstructural features in annealed metals and ceramics have shown non-uniform distributions of internal interfaces [5–13]. In each case, these distributions appear to be related to the physical properties of the interfaces. At the same time, several alloy producers are now advertising materials which have undergone thermal and mechanical processing to control the frequency of "special" internal interfaces. Similar processes are discussed in the literature, and have been shown to result in improved bulk properties such as resistance to intergranular fracture and stress corrosion cracking [14–19], longer fatigue lifetimes [20, 21], or enhanced ductility [22–24].

The question of how such non-uniform distributions of internal interfaces form has not yet been answered. Grain growth in a single-phase polycrystal with anisotropic interfacial properties is perhaps the simplest physical process that might lead to interface texture, and several computational studies have been performed that attempt to relate anisotropy to interface texture development in these systems [1, 25–38]. Results from these simulations suggest that grain growth with anisotropic interfacial properties can result in grain boundary distributions similar to those measured in real materials. None of the previous work, however, incorporates complete interfacial anisotropy or fully three-dimensional grain growth. While quantitative models have been proposed that explain area-weighted grain boundary statistics [28], no such model exists that predicts texture in both area- and number-weighted grain boundary distributions, as observed in experiment [6]. There is then a need to investigate interface texture development using more complete models.

1.2 Objective

The objective of this work is to determine the effect of anisotropic interfacial properties on interface texture resulting from grain growth. Specifically, we study the the misorientation distribution function (MDF) and the grain boundary character distribution (GBCD), both statistical measures of the interface character of a polycrystal. Our focus is on determining the mechanisms which produce interface texture and developing a quantitative model that relates interfacial properties to the MDF or GBCD.

1.3 Hypotheses

We predict that many of the observations from previous simulations and experiments will hold in the present case, i.e. that anisotropic interfacial properties will result in non-uniform interface distributions, that these distributions will generally exhibit an inverse relation to interfacial energy, and that mobility anisotropy will have a weaker effect on grain boundary distributions than energy anisotropy [31, 32, 37]. In microstructures with random orientation texture, we assume that the process of grain growth introduces new grain boundaries randomly, and because the kinetic equation of grain growth is independent of length scales, we expect to find that the GBCD and MDF show steady-state behavior during grain growth in sufficiently large (many grains) systems. Because we expect texture development in our simulations to result from anisotropic interfacial properties, we predict that the amplitudes in most distributions increase with increasing energy anisotropy. To test these hypotheses we must be able to measure grain boundary character in systems with varying levels of anisotropy, while also analyzing interface texture data at many sequential points in time.

1.4 Method

Experimental measurements of the MDF are common, and GBCD measurements have been performed recently [5, 6, 8, 9]. While automated texture analysis systems are continually improving, such measurements still require a non-trivial effort to produce a temporal sequence of distributions to a desirable resolution and accuracy. Additionally, those interfacial properties (energy and mobility) that affect grain growth kinetics are not easily measured. To study the effect of grain growth with anisotropic interfacial properties on the MDF or GBCD, we must first be able to associate an interfacial energy and mobility with any given misorientation and interface plane. At this time, only one energy measurement with this generality has been completed, and there has never been such a mobility measurement [4]. Our objective is perhaps unattainable with a purely experimental approach.

We then proceed with computational methods. Mesoscale grain growth simulations have already been used to successfully model physical systems with anisotropic interfacial properties [1,25,27–29, 33,34,36]. Measurement of the MDF or GBCD are then trivial, and the desired interfacial anisotropy becomes an input parameter. While we focus on simulations that have realistic material properties, using computational methods also allows us to model systems with interfacial properties that may not be easily achieved with a physical specimen. For example, we can easily scale energy and mobility functions, or isolate the effects of either. Exaggerated anisotropies can be easily simulated and could be used to test the robustness of our hypotheses.

The variablity of the systems that might be studied is enormous. Our approach is to work from the ground up, working with the simplest relevant cases. We therefore impose the following limits on our model systems:

1. All systems are single-phase polycrystals. These are considered to be pure materials, i.e. there are no effects of strutural or chemical inhomogeneities such as solute segregation that might

affect the kinetics of grain growth.

2. In all systems, the correlated ODF and the uncorrelated ODF are, for practical purposes, the same. This means that the statistical properties of the microstructure are the same in either a local or global sense.

3. Thermodynamic variables other than interfacial energy (temperature, pressure, etc.) are constant and uniformly homogeneous. In particular, the bulk free energy density is constant, which implies that the only driving force for grain growth is the reduction in interfacial energy.

4. Interfacial energy and mobility are functions of macroscopic interface geometry only.

We note that the methods used in this work could easily be applied to systems where any of the above limits are relaxed, and such cases should be examined in future work.

Chapter 2

Background

This chapter reviews basic concepts used in our work. Crystallography and polycrystallography are discussed in the first section. The characterization of polycrystals by their macroscopic geometry provides the most convenient method of classifying internal interfaces. The second section introduces the interface texture measures that will be the objects of our study. These measures are statistical distributions of internal interfaces based on the classification method discussed earlier. The following two sections cover interfacial properties and the kinetic equation of grain growth, respectively. In particular, we discuss grain boundary energy and mobility, which are the essential input to our models. We then present an overview of methods commonly used in grain growth simulation. Finally, we examine previous experimental and computational results that relate grain boundary properties to interface texture development.

2.1 Characterization of internal interfaces

2.1.1 Crystallography

Crystallography is the study of crystal structures, which are spatially periodic arrangements of atoms or molecules in space [39]. By convention, we associate coordinate axes with some convenient directions in a crystal and describe features, such as directions or atomic planes, using the terminology of vector space theory. This model applies to a wide range of metals, alloys, ceramics, and naturally

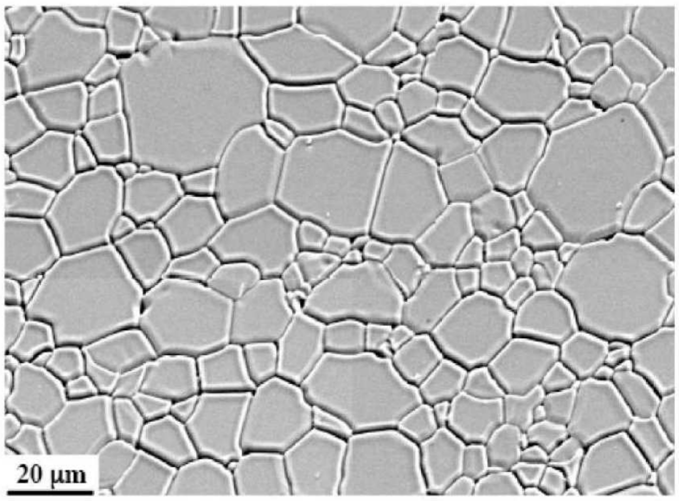


Figure 2.1: Scanning electron microscope image of strontium titanate microstructure (Courtesy T. Sano).

occuring minerals.

Crystal structures are often classified by their symmetry, or more precisely, by invariance under certain unitary (length and angle preserving) linear transformations. Such transformations represent operations such as rotating a crystal from one spatial orientation into a physically indistinguishable orientation. Symmetry implies that the directionally dependent properties of a crystal are invariant under the same transformations (see e.g. [40]).

2.1.2 Polycrystallography

A polycrystal is a collection of crystals or grains that make up a single solid object, figure 2.1. Individual grains in a polycrystal may differ from their neighbors by either orientation or phase. Most engineered alloys and ceramics are polycrystalline.

The orientations of individual grains are usually represented mathematically as proper rotations from some global coordinate system. Many computational tools exist for characterizing rotations (see e.g. [41]). For generating orientations, and in functional arguments, we use Bunge's convention [42] for Euler angles, written as φ_1, Φ, and φ_2. For descriptive purposes, we will often specify particular orientations by their associated rotation axis (unique real eigenvector) and angle pair. We use the quaternion representation exclusively for computations [43].

The relative orientation difference between two grains, or the misorientation, is an important quantity that provides one means for characterizing a grain boundary. The misorientation is itself a rotation and is parameterized as such. Locally, the interface between a pair of adjacent grains is represented as a two-dimensional surface. The local normal vector to the surface can be represented by spherical angles with respect to the coordinate axes of either grain. In general, the macroscopic geometric character of a grain boundary is determined by the misorientation between the opposing grains and the direction of the local interface normal vector.

In three-dimensional polycrystals, any orientation or misorientation has three mathematical degrees of freedom [41]. Another related classification scheme for grain boundary character compresses misorientations into a single parameter called the disorientation angle. The disorientation angle of two grains is the smallest possible rotation angle when misorientation is described as one of any symmetrically equivalent rotation axis and angle pairs. Grain boundaries with similar disorientation are sometimes found to have similar physical properties, and so the disorientation classification is used frequently.

Some grain misorientations have especially simple geometries, and have been studied extensively in the literature. These include twins and other coincident site lattice (CSL) boundaries [44]. These interfaces have some number of lattice sites on either side of the boundary plane which coincide. In some cases, CSL boundaries have physical properties that are significantly different than arbitrary grain boundaries. The atomic structures and physical properties of more general grain boundaries are in most cases unknown [45].

2.2 Interface texture measures

Polycrystalline materials are rarely simple enough to allow a complete quantitative characterization of all microstructural features. Statistical measures, such as those described below, have proven useful in understanding and predicting materials behavior.

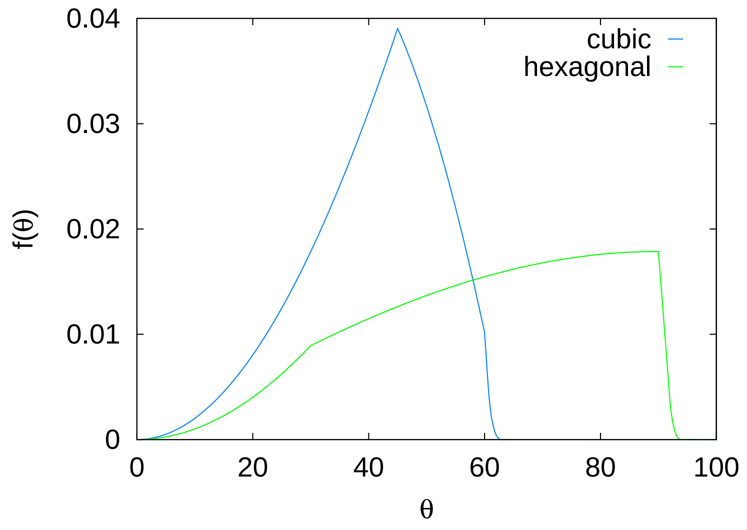


Figure 2.2: Random disorientation-based MDFs for cubic (point group O) and hexagonal (point group D6) crystal systems.

2.2.1 Misorientation distribution function

A misorientation distribution function (MDF) quantifies the probability of measuring interfacial area in a polycrystal where the two grains making the interface have crystal coordinate systems related by a particular rotation. The MDF can be interpreted as a weighted probability density on the group of rotations in three dimensions. The weighting factor is usually chosen such that a random distribution has unit value everywhere, i.e. the value is given in multiples of the random distribution (MRD).

An MDF has symmetry properties derived from the crystal system(s) present in the microstructure. Often, the MDF is compressed into a probability density based on disorientation (see figure 2.2). Because the misorientation is constant at all points on an interface separating two grains, an MDF may in some cases be used to express a number fraction of boundary types rather than an area fraction. How the MDF is used will always be stated explicitly.

In all that follows, the MDF will be presented as a function of disorientation angle θ only. The MDF is computed by discrete binning. We consistently use a total of 30 bins over a range of disorientation angles $2° \leq \theta \leq \theta_{\max}$, where $\theta_{\max}$ is the maximum disorientation angle for the given

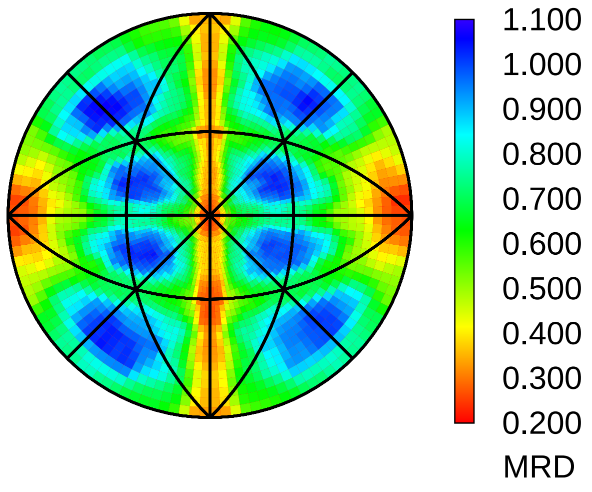


Figure 2.3: Example GBCD from simulation (see chapter 5) [1]. Population (in multiples random) of grain boundary planes at a fixed misorientation of 45° about $\langle 1, 0, 0 \rangle$.

crystal system. For cubic and hexagonal crystal systems, we use the approximations $\theta_{\max} = 62.8°$ and $\theta_{\max} = 94.0°$, respectively.

2.2.2 Grain boundary character distribution

The grain boundary character distribution (GBCD) incorporates both misorientation and interface normal vector information into a single weighted probability density.

The GBCD is a function of the five parameters describing misorientation and the interface normal. Because of the possibility of continuously changing interface normal vectors, the GBCD cannot be expressed as a number fraction of boundary types as the MDF without some explicit choice of discretization. The convention for visualizing the GBCD is to plot sections of constant misorientation in a stereographic projection as shown in figure 2.3. Experimental measurements of the GBCD have been performed only recently [5, 6, 8, 9].

Like the MDF, we compute the GBCD by discrete binning. We use an equal volume element in the space of grain boundary planes with an approximate resolution of 10°, i.e. $\Delta\phi_1 = \Delta\phi_2 = \Delta\phi = 10°$ and $\Delta \cos \Phi = \Delta \cos \theta = 1/9$. Because we compute the GBCD only for the cubic crystal system, we use a reduced Euler space defined by $0° \leq \phi_1 \leq 90°$, $0° \leq \Phi \leq 90°$, and $0° \leq \phi_2 \leq 90°$, and the positive hemisphere of S^2, i.e. $0° \leq \phi \leq 360°$ and $0° \leq \theta \leq 90°$.

2.3 Physical properties of internal interfaces

Here we discuss those properties of internal interfaces which are explicitly involved in grain growth kinetics. The discussion here should make it clear that, although a large number of simplifying approximations must be made, the energy and mobility functions used in our simulations do have an underlying physical basis.

2.3.1 Energy

Internal interfaces such as grain boundaries are non-equilibrium defects which are introduced into a material through processing [45]. It follows that there is a positive excess energy associated with such interfaces. This energy depends on interface structure as well as thermodynamic variables such as temperature and chemical composition. The present work only considers systems where temperature and other thermodynamic parameters are constant, and therefore interfacial energy will be a function of interface geometry alone.

Even ignoring thermodynamic parameters, grain boundary energy as a function of macroscopic interface geometry is known for relatively few materials systems [4]. It is therefore common practice to approximate boundary properties by some simple functional forms. For misorientation dependence, the most common approximation by far is the relation first proposed by Read and Shockley [46]. The Read-Shockley energy γ is a function of disorientation θ and takes the form

$$\gamma(\theta) = \begin{cases} \gamma' \frac{\theta}{\theta'} \left[1 - \ln\left(\frac{\theta}{\theta'}\right)\right] & \theta < \theta' \\ \gamma' & \theta \geq \theta' \end{cases} \tag{2.1}$$

where γ' is the average energy of high angle grain boundaries and θ' is the high disorientation angle cutoff. A typical choice of θ' is 15°, and this function is shown in figure 2.4. Note that the use of the

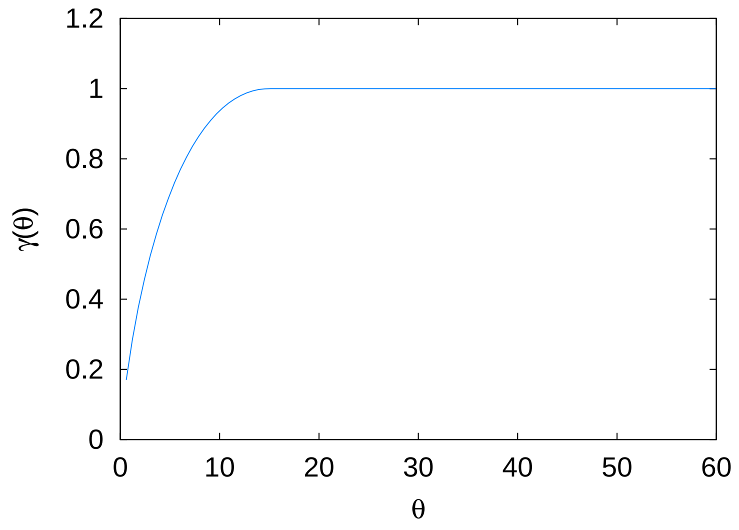


Figure 2.4: Read-Shockley energy plotted with $\theta' = 15°$.

disorientation angle implies that this function of grain boundary energy is invariant under all proper crystal symmetry operators. The Read-Shockley function is based on a model of grain boundaries as arrays of dislocations, and has been shown to fit energy functions for various polycrystalline systems in the low angle regime [47–49]. The expression given above ignores differences in high angle grain boundary energies where the dislocation array model is no longer appropriate. For example, Hasson et al. have shown that the deviations occur at high angle tilt boundaries with low index planes in aluminum and copper [2]. Their data is reproduced in figure 2.5. However, such deviations are relatively small and the Read-Shockley function provides a reasonable approximation. Especially large deviations from the Read-Shockley energy are expected to occur near particular misorientations such as coherent twins or coincident site lattice (CSL) boundaries [50, 51].

The dependence of grain boundary energy on both misorientation and interface plane normal has been studied recently. Saylor et al. [4] have shown that a half-energy value can be associated with any interface plane irrespective of misorientation, such that the inclination dependent grain boundary energy can be approximated reasonably well by the sum of the half-energies associated with the two interface planes, i.e.

$$\gamma(\Delta\mathbf{g}, \mathbf{n}_A) = \gamma_s(\mathbf{n}_A) + \gamma_s(\Delta\mathbf{g}\mathbf{n}_A) \tag{2.2}$$

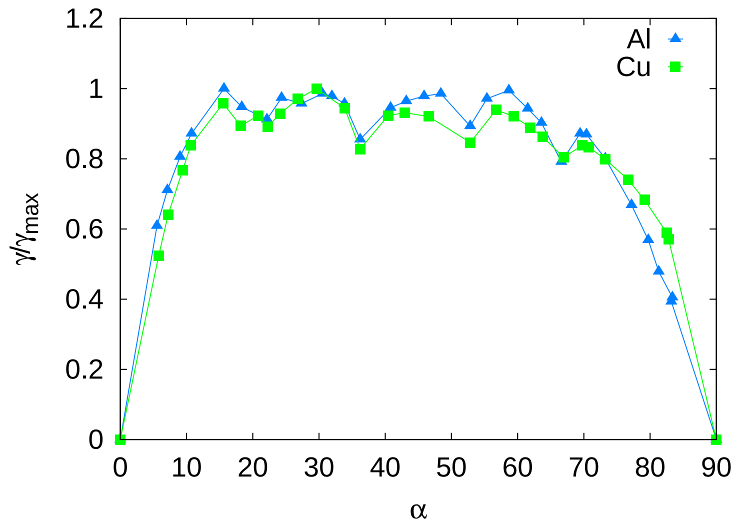


Figure 2.5: Energy of symmetric tilt grain boundaries in aluminum and copper as a function of tilt angle α about [100] from Hasson et al. [2].

where $\mathbf{n}_A$ denotes the interface normals in one of the local (grain) coordinate systems, and $\Delta\mathbf{g}$ is a linear operator representing the grain misorientation. The function γ_s approximates the surface energy of a single crystal. This method for assigning grain boundary energy with inclination dependence has been shown to produce simulated GBCDs which in large part agree with experimental results [1].

2.3.2 Mobility

Interface mobility is defined as the ratio of the normal velocity of a moving interface to the driving force for that motion (see the following section). For the systems studied here, mobility is considered only as a function of interface geometry only, similar to the interfacial energy.

A reasonable approximation for grain boundary mobility as a function of misorientation in many materials is given by [52]

$$M(\theta) = \begin{cases} M' \left(\frac{\theta}{\theta'}\right)^n & \theta < \theta' \\ M' & \theta \geq \theta' \end{cases} \tag{2.3}$$

where M' is the average mobility of high angle grain boundaries, n is an empirical constant, and

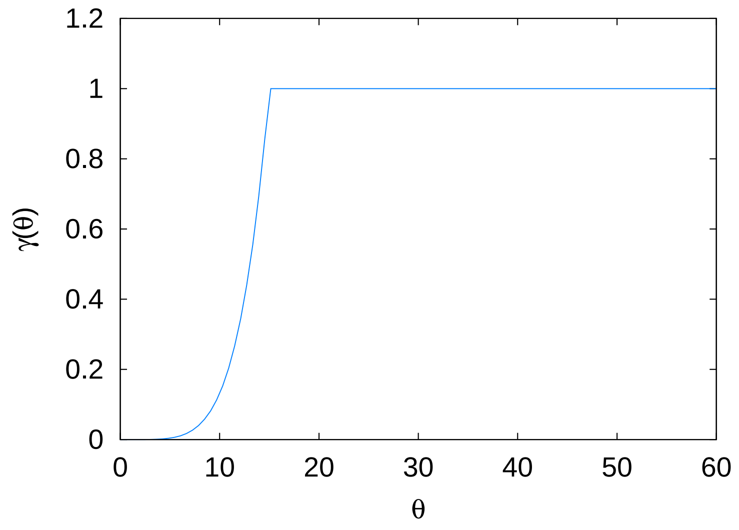


Figure 2.6: Mobility as a function of misorientation, given in equation 2.3. Here $n = 5$.

θ' is the high angle disorientation cutoff (figure 2.6). Typically, the value of n is chosen to be an integer greater than one. Qualitatively, this function behaves like the Read-Shockley function, with a constant value for high angle boundaries and a fast drop to zero at zero misorientation. The misorientation dependence of mobility is often approximated by either expression. As in the approximation for grain boundary energy, this relation does not incorporate differences in high angle boundary mobilities, which are known to change rapidly near special boundaries [53].

No complete experimental measurements of grain boundary mobility as a function of both misorientation and interface plane normal have been performed as of this time. However, it seems reasonable to chose an inclination dependent mobility function in the same way as described above. Several measurements of mobility for specific grain boundaries have shown that anisotropies as great as 100 or larger can occur in the space of grain boundary planes [53]. These are typically associated with CSLs or other special boundaries.

2.4 Kinetics of grain growth

Grain growth is the process by which the total energy in a polycrystal is reduced by the elimination of internal interfacial area as grain boundaries move. This motion is a result of solid-state diffusion between adjacent grains. If a volume exchange that occurs at an interface decreases total energy, the process leads to the growth of one grain at the expense of the other. We now relate the energy change through interface motion to velocity for grain boundaries.

The simple model given here follows Turnbull [54]. At high temperatures, solid-state diffusion occurs and there is a diffusive flux of atoms from grain 1 to grain 2, as well as in the opposite direction. As a diffusional process, each flux is proportional to an exponential function of the activation energy required for atomic motion. The net flux from grain 1 to grain 2 is

$$J = c_{12} \exp\left[-\frac{\Delta E_{12}}{kT}\right] - c_{21} \exp\left[-\frac{\Delta E_{21}}{kT}\right] \tag{2.4}$$

where each c_{ij} is a positive constant, T is temperature, k is Boltzmann's constant, and ΔE_{ij} is the activation energy required for flux from grain i to grain j. Define $\Delta E = \Delta E_{12} - \Delta E_{21}$. When $\Delta E = 0$, there is no change in energy related with interface motion, and therefore $J = 0$, which implies that the proportionality constants $c_{12} = -c_{21} = c$. Assuming $J > 0$, at high temperatures $kT \gg \Delta E$ and, keeping only the linear terms of the exponential,

$$J \approx c \exp\left[-\frac{\Delta E_{12}}{kT}\right] \frac{\Delta E}{kT} \tag{2.5}$$

The normal velocity of the interface v_n is given by the product of the net diffusive flux and the atomic volume N_a/V_m, giving

$$v_n = M \frac{\Delta E}{\Delta V_m} \tag{2.6}$$

where the interface mobility M is introduced as the proportionality between the energetic driving force for boundary motion and the interface velocity.

We now have an equation which relates the local change in energy due to interface motion to the velocity of an internal interface. In the continuum limit, this equation represents boundary motion with velocity proportional to the energy gradient. For smoothly curved surfaces, this relation can be expressed as

$$v_n = M\gamma\kappa \tag{2.7}$$

where κ is mean curvature. This equation now explicitly incorporates grain boundary energy. Note that the above derivation does not consider other mechanisms for grain boundary motion, such as grain sliding or cooperative atomic motion [55].

2.5 Grain growth simulation

Several methods for simulating macroscopic grain growth are in common use and are discussed in this section. More detailed information about the methods used in this work is presented in the next chapter.

Vertex method. The vertex method has been used mainly to simulate grain growth in two dimensions [56–66]. This method tracks triple junctions, or vertices, and approximates the remainder of the grain boundary as a line segment (2D) or a planar segment (3D). Forces on the vertices are calculated by the force imbalance at the triple junctions as per Herring's equation for triple junction equilibrium [67], and the system is evolved through discrete time steps.

Relatively few vertex method simulations with anisotropic interfacial properties have been performed [57, 60]. The vertex method cannot accurately predict the shape of curved grain boundaries, and so is probably of limited use for studying local interface character and related texture measures such as the GBCD.

Front-tracking methods. Front-tracking methods[1] are essentially extensions of the vertex method. In addition to the motion of triple junctions, the motion of arbitrary points along the interface are calculated. The earliest such method was two-dimensional and used the points between triple junctions to interpolate curved interfaces [68]. Triple junction motion was then similar to that in the vertex method but a separate step was taken to advance all non-vertex points by interface curvature. Many 2D front-tracking methods take this form [68–72], although particular methods may use some definite choice of interpolating curve, e.g. cubic splines [73, 74].

Front-tracking methods improve upon the simple vertex method in that more realistic interface curvatures are possible. Thus it is feasible to use such methods to study local interface character.

[1]Often, the term "front-tracking method" is used interchangably with "sharp interface method." With this terminology, moving finite element (MFE) methods are front-tracking methods. However, we will make the distinction that front-tracking methods are those not derived from a variational principle.

Monte Carlo methods. Monte Carlo (MC) methods are perhaps the most commonly used methods for simulating grain growth [28, 34, 75–95]. The basis of MC is that the physical domain is partitioned into identical cells, each with an associated state which represents orientation, phase, or other properties. Grain growth occurs by randomly selecting cells and altering their state by some probabalistic rule. The probability is chosen such that the system evolves as if by a thermally activated process. The MC method naturally lends itself toward simulating anisotropic properties, e.g. in simulating abnormal grain growth due to anisotropic mobility [78, 81, 82].

Cellular automata methods. Cellular automata (CA) methods, like Monte Carlo methods, are simulations in a spatial domain of regular cells with discrete states. In classical CA methods, the new state of each cell in a CA simulation is updated by a deterministic rule that depends only on the previous state of the cell and the neighborhood of cells around it. Another approach is to use a probabilistic rule like that used for MC [96]. Such CA methods still differ from MC methods in that cell updates are performed simultaneously.

Several grain growth simulations with CA methods have been performed [96–101]. An emerging approach to grain growth simulation with CA is to use cells defined by points randomly positioned within the simulation domain. Such random-grid CA models have even been used to simulate anisotropic grain growth [98].

Phase field methods. Phase field methods for simulating grain growth were developed later than most of the methods mentioned above [30–32, 102–112]. In phase field (PF) methods, continuous fields are defined over the simulation domain and represent individual phases or orientations. Interfaces are represented by level sets of the phase fields and are moved implicitly by evolving the fields.

PF simulations with anisotropic interfacial properties have been performed, many incorporating interface plane anisotropy [30–32, 108–110]. Until recently, the PF method has been capable of simulating grain growth only in two-dimensional or small three-dimensional systems. New computational methods have overcome these limitations and made the PF method suitable for large-scale, three-dimensional simulations with anisotropy [112, 113].

Moving finite element methods. Finite element methods (FEM) have many well-known applications. For simulating grain growth, moving finite elements (MFE) are employed [1, 29, 33, 35, 37, 114–119]. MFE methods represent interfaces as discrete elements, and boundary motion occurs by updating the elemental node positions. The node velocities are calculated as the solution to some variational problem and positions are updated through discrete time steps.

Moving finite element methods for simulating grain growth use a discretization in which a wide range of interface inclinations can be approximated quite naturally. MFE methods make it trivial to assign anisotropic interfacial properties which are interface plane dependent, to incorporate these properties into the numerical method, and to analyze results.

2.6 Anisotropy and texture

The effect of anisotropic interfacial properties on interface texture development has been studied previously with Monte Carlo and Phase Field methods [25, 27, 28]. In addition, Kinderlehrer et al. have performed 2D moving finite element simulations to study interface texture [35].

Hinz et al. [25] assigned low energy values to coincident site lattice (CSL) boundaries and observed higher populations of such grain boundaries in three-dimensional MC simulations. In three similar studies, 2D MC simulations of grain growth with Read-Shockley type interfacial energy produced interface populations with increased frequency of low-angle (low energy) boundaries [27, 28, 34]. The work of Kazaryan et al. [30–32, 110] incorporated anisotropic energy and mobility dependent on both misorientation and interface inclination into 2D PF models. Results from these simulations suggested a relatively weak effect of anisotropic mobility in comparison to anisotropic energy. Anisotropic energy was found to increase the population of low energy interfaces. Upmanyu et al. [120] performed similar calculations with misorientation-dependent energy and mobility [9]. Again, the effects of anisotropic energy on interface plane texture were significantly greater than those of anisotropic mobility. The authors of this paper proposed a boundary lengthening mechanism based on triple junction equilibrium to explain their results. Kinderlehrer et al. [35] have simulated grain growth with misorientation and interface plane dependent energy and mobility using a 2D FEM model. Their results largely confirm those described above. In addition, they simulated grain growth with anisotropic properties while constraining triple junctions to meet isotropic equilibrium conditions.

The result was no interface texture development, supporting the idea that triple junctions influence the development of interface plane populations. The authors also suggest interface rotation at triple junctions as a cause of texture in the space of interface planes.

Chapter 3

Simulation methods

This chapter contains detailed information about the numerical methods used in this work. The essential features of each simulation method are discussed, with particular emphasis on the incorporation of interfacial anisotropy. Test cases are presented which demonstrate that the methods perform as expected for a variety of simple geometries. Finally, microstructure generation and analysis methods are discussed.

Most of the computations discussed in the following chapters were performed with software written specifically for this project. The appendix contains a detailed listing of the most relevant source code.

3.1 Monte Carlo method

The formulation of the Monte Carlo method used here is based largely on common practice (see e.g. [28,81,93,121]). We consider a domain Ω, sampled on a Cartesian grid with N lattice sites. Each site has an integer value associated with it, called its "spin" state. For our purposes, the spin represents a grain identifier, or equivalently, a fixed crystal orientation. In this representation, adjacent sites with unlike spins define grain boundaries.

The total system energy is given by

$$E = \sum_{i=0}^{N} \sum_{j \neq i}^{N} \gamma(\theta_{ij}) \chi(i,j) \tag{3.1}$$

where N is the number of lattice points, $\gamma(\theta_{ij})$ is the energy per unit area of grain boundary with misorientation θ_{ij}, and $\chi(i,j)$ is a function that is one if site i is a near neighbor of site j and zero otherwise. For a general lattice site, we use the 8 (2D) or 26 (3D) nearest lattice sites as neighbors. Finite boundary conditions are imposed, and therefore sites on the domain boundary have fewer neighbors than interior sites.

The Monte Carlo algorithm begins by choosing a site at random, which we will call the base site. The state, or "spin" of the base site is recorded. Then a list is generated that consists of spins found in the base site's nearest neighbors. A possible new spin for the base site is chosen randomly from this list. Define ΔE, the change in E produced by replacing the base site's old spin with the new spin. The switch is allowed to occur with a probability P that depends on the total energy change of the system, as well as grain boundary energy and mobility,

$$P = \begin{cases} \frac{M(\theta_{ij})}{M_{\max}} \frac{\gamma(\theta_{ij})}{\gamma_{\max}} & \Delta E \leq 0 \\ \frac{M(\theta_{ij})}{M_{\max}} \frac{\gamma(\theta_{ij})}{\gamma_{\max}} \exp\left[-\frac{\Delta E}{\beta \gamma(\theta_{ij})}\right] & \Delta E > 0 \end{cases} \tag{3.2}$$

where β is an empirical constant chosen such that interfaces roughen but do not disorder. In most simulations, we use values of β either 0.7 (2D) or 1.5 (3D). Intuitively, P represents the probability that the subsystem at the base site has sufficient thermal energy to make the switch. The process is then repeated. We use the convention that N such repetitions of this process constitute one Monte Carlo step (MCS).

Note that anisotropy has been included explicitly in the flip probability given by equation 3.2. While both the interface mobility and energy scale the magnitude of the boundary migration rate, interfacial energy also affects the driving force for migration through the computation of ΔE with equation 3.1. Thus energy and mobility anisotropy perform distinct roles in Monte Carlo grain growth simulations.

Several choices made here, while common, are only a matter of preference when performing

Monte Carlo simulations. For example, we perform Monte Carlo simulations on Cartesian grids, although other discretizations such as the triangular lattice (2D) are common. Likewise, periodic boundary conditions are often used as an alternative to finite boundary conditions. While these choices may have little effect on the physicality of the simulation, it is known that choosing an inappropriate value of β can lead to lattice pinning (low β) or disordering (high β). In this work, we will demonstrate that all of the relevant data we wish to extract from Monte Carlo simulations are fairly insensitive to the boundary conditions used or to our choice of lattice temperature β.

3.2 Moving finite element method

The classical equation for interface-controlled boundary motion is

$$v_n = M \nabla_n E \tag{3.3}$$

where v_n denotes the magnitude of the interface normal velocity, M the interface mobility, and $\nabla_n E$ the total system energy change by an infinitesimal normal displacement of the interface at a given point. The direction of the interface normal is, of course, chosen so that the process is energy dissipative. The moving finite element method attempts to move mesh nodes so that the equation of motion is satisfied as well as possible at all points on each interface [114–116]. This naturally leads to minimization of a functional such as

$$J_1 = \int_S (v_n - M \nabla_n E)^2 \, dS \tag{3.4}$$

over all possible nodal velocities. Here S implies all mobile interfaces. This functional represents in some sense a least-squares problem for calculating interface motion.

Two additional issues must be addressed by the method. First, minimization of (3.4) may not produce unique solutions for some mesh geometries. As an example, consider a planar interface with a constant velocity at all points, presumably moving by a homogeneous volume energy driving force. Minimization of (3.4) ensures that the boundary nodes will move with equal velocities perpendicular to the interface, but does not constrain the lateral motion of any such nodes. Secondly, it is desirable that all nodes move so as to maintain a consistent mesh size and aspect ratio of volume elements. This is critical in applications where field variables are calculated at the nodes. We must therefore make an appropriate modification to the functional.

Each of the above problems may be addressed by controlling the size and quality (aspect ratio) of interface and volume elements in the mesh. Intuitively, we would like to incorporate additional forces on the nodes so that their motion best maintains mesh element quality. This leads to another functional of the form

$$J_2 = \epsilon^{tri} \sum_{nodes} \left\| \dot{\mathbf{x}}^{(i)} - M^{tri} \frac{\partial Q^{tri}}{\partial \mathbf{x}^{(i)}} \right\|_2^2 + \epsilon^{tet} \sum_{nodes} \left\| \dot{\mathbf{x}}^{(i)} - M^{tet} \frac{\partial Q^{tet}}{\partial \mathbf{x}^{(i)}} \right\|_2^2 . \tag{3.5}$$

Here, the ϵ and M terms are positive constants and the ∇Q terms represent gradients in artificial "energies" associated with element size and aspect ratio. These will be discussed in more detail later. The importance assigned to interface and volume element quality relative to the physically meaningful forces acting on the interface can be controlled independently by the ϵ parameters. The magnitude of these parameters is always chosen such that $J_2 \ll J_1$, i.e. so that the natural forces on the boundary are dominant. The M parameters determine the relative mobility of nodes under the applied element quality forces. In this formulation, each M may be chosen independently but elements of the same type are treated equally (isotropic mobility).

The complete moving finite element method is then to minimize the functional

$$J = J_1 + J_2 \tag{3.6}$$

with respect to the motion of all mesh nodes. Let ϕ_i denote a piecewise linear basis function which is unity at node i and zero at all other nodes. The j^{th} position vector component x_j of any other point within the mesh can then be written as

$$x_j = \sum_{i=1}^{N} \phi_i x_j^{(i)} \tag{3.7}$$

where $x_j^{(i)}$ is the j^{th} component of the position vector of the i^{th} node, and N is the total number of mesh nodes. The j^{th} component of velocity $\dot{\mathbf{x}}_j$ at any point is found by differentiation of (3.7) with respect to time,

$$\dot{\mathbf{x}}_j = \frac{\partial}{\partial t}\left[\sum_{i=1}^{N} \phi_i x_j^{(i)}\right] = \sum_{i=1}^{N} \phi_i \dot{x}_j^{(i)}. \tag{3.8}$$

For any vector w,

$$\|\dot{\mathbf{x}} - w\|_2^2 = \sum_{j=1}^{3}\sum_{i=1}^{N}\left[\phi_i \dot{x}_j^{(i)} - w_j\right]^2. \tag{3.9}$$

The magnitude of the normal velocity at any point on an interface is found by projection along the (local) interface normal vector,

$$v_n = \dot{\mathbf{x}} \cdot \mathbf{n} = \sum_{j=1}^{3}\sum_{i=1}^{N} \phi_i \dot{x}_j^{(i)} n_j. \tag{3.10}$$

It is now possible to rewrite the functional (3.6) with nodal velocities as the independent parameters. For simplicity, we will address each integral separately. The integral (3.4) expands to

$$\int_S \left[\sum_{j=1}^{3}\sum_{i=1}^{N} \phi_i \dot{x}_j^{(i)} n_j - M \nabla_n E\right]^2 dS. \tag{3.11}$$

Similarly, substitution of (3.9) into (3.5) gives

$$\epsilon^{tri} \sum_{j=1}^{3}\sum_{i=1}^{N}\left[\phi_i \dot{x}_j^{(i)} - M^{tri} \frac{\partial Q^{tri}}{\partial x_j^{(i)}}\right]^2 \tag{3.12}$$

and

$$\epsilon^{tet} \sum_{j=1}^{3}\sum_{i=1}^{N}\left[\phi_i \dot{x}_j^{(i)} - M^{tet} \frac{\partial Q^{tet}}{\partial x_j^{(i)}}\right]^2 \tag{3.13}$$

respectively.

A necessary condition for a minimum of (3.6) is that its derivatives with respect to the $3N$ nodal velocity components are zero. Differentiation gives

$$\begin{aligned} \frac{\partial J}{\partial x_k^{(h)}} = &\quad 2\int_S \left[\sum_{j=1}^{3}\sum_{i=1}^{N} \phi_i \dot{x}_j^{(i)} n_j - M \nabla_n E\right] \phi_h n_k \, dS \\ &+ \quad 2\epsilon^{tri}\left[\dot{x}_k^{(h)} - M^{tri}\frac{\partial Q^{tri}}{\partial x_k^{(h)}}\right] + 2\epsilon^{tet}\left[\dot{x}_k^{(h)} - M^{tet}\frac{\partial Q^{tet}}{\partial x_k^{(h)}}\right] = 0 \end{aligned} \tag{3.14}$$

which must be satisfied for $1 \leq h \leq N$ and $1 \leq k \leq 3$. Combining terms and reordering,

$$\sum_{j=1}^{3}\sum_{i=1}^{N}\left[\int_S n_j n_k \phi_i \phi_h \, dS + (\epsilon^{tri} + \epsilon^{tet})\delta_{ih}\delta_{jk}\right]\dot{x}_j^{(i)} =$$
$$\int_S \phi_h n_k M \nabla_n E \, dS + \epsilon^{tri} M^{tri} \frac{\partial Q^{tri}}{\partial x_k^{(h)}} + \epsilon^{tet} M^{tet} \frac{\partial Q^{tet}}{\partial x_k^{(h)}} \qquad (3.15)$$

This is a system of $3N$ ordinary differential equations, which can be solved by a number of methods [122]. For simplicity, we will express the previous equation as

$$\mathbf{K}\dot{x} = \mathbf{F}. \qquad (3.16)$$

Note that the matrix $\mathbf{K}$ contains only geometric information about the mesh. Although other choices for the element quality forces (equation 3.5) might be more robust [116], in this case the element quality contribution is simply the diagonal matrix $(\epsilon^{tri} + \epsilon^{tet})I$, which leads to faster numerical convergence.

We now examine the process of evaluating the integrals appearing in equation 3.15. Note that, in practice, it is sufficient to loop through each surface element, performing the following calculations only once while assembling the global stiffness matrix $\mathbf{K}$ and force vector $\mathbf{F}$. The left-hand side surface integral is easily calculated by a summation of their value on individual surface (triangle) element domains. Since each triangle has a unique normal vector, the normal components are constants on every domain of integration. Therefore, it is enough that we demonstrate how to calculate

$$\int_S \phi_i \phi_h \, dS \qquad (3.17)$$

for an arbitrary triangular element.

A parent triangular element can be defined with vertices at $(0,0)$, $(1,0)$, and $(0,1)$ in a coordinate system parameterized by u and v. We will map the vertices $(1,0)$ and $(0,1)$ to the nodes $x^{(i)}$ and $x^{(h)}$, respectively. The remaining node will be labeled $x^{(0)}$. In this coordinate system, the functions $\phi_i = u$ and $\phi_h = v$. The coordinate mapping $x_j(u,v)$ from the parent triangle to any mesh triangle is given by

$$x_j(u,v) = (x_j^{(i)} - x_j^{(0)})u + (x_j^{(h)} - x_j^{(0)})v + x_j^{(0)} \qquad (3.18)$$

and so

$$\begin{aligned} x_u(u,v) &= \frac{\partial}{\partial u}x(u,v) = \langle x_1^{(i)} - x_1^{(0)}, x_2^{(i)} - x_2^{(0)}, x_3^{(i)} - x_3^{(0)} \rangle \\ x_v(u,v) &= \frac{\partial}{\partial v}x(u,v) = \langle x_1^{(h)} - x_1^{(0)}, x_2^{(h)} - x_2^{(0)}, x_3^{(h)} - x_3^{(0)} \rangle. \end{aligned} \tag{3.19}$$

The metric tensor $\mathbf{g}$ is therefore constant and, since $\sqrt{\mathbf{g}} = 2A$,

$$\int_S \phi_i \phi_h \, dS = \int_0^1 \int_0^{1-v} uv\sqrt{g}\, du\, dv = \frac{1}{12}A. \tag{3.20}$$

Here A is the area of the particular triangular element.

For the right-hand side surface integral, we again reduce the integral into a summation of its value on individual triangular elements. In this case, interfacial mobility M and n_k are constant. It is then sufficient to calculate

$$\int_S \phi_h \nabla_n E \, dS \tag{3.21}$$

for an arbitrary interface triangle. First note that from equation 3.7 it follows that

$$\frac{\partial x_r}{\partial x_r^{(h)}} = \phi_h. \tag{3.22}$$

for $1 \le r \le 3$. At any point on a triangular element

$$\nabla_n E \equiv \sum_{r=1}^{3} \frac{\partial E}{\partial x_r} n_r. \tag{3.23}$$

Then

$$\begin{aligned} \int_S \phi_h \nabla_n E \, dS &= \int_S \sum_{r=1}^{3} \frac{\partial x_r}{\partial x_r^{(h)}} \frac{\partial E}{\partial x_r} n_r \, dS \\ &= \int_S dS \sum_{r=1}^{3} \frac{\partial E}{\partial x_r^{(h)}} n_r = A \sum_{r=1}^{3} \frac{\partial E}{\partial x_r^{(h)}} n_r \end{aligned} \tag{3.24}$$

since each $\partial E / \partial x_r^{(h)}$ term is constant. Note that, for a triangular element, $E = \gamma A$. We compute $\partial E / \partial x_r^{(h)}$ by numerical perturbation. The perturbed $(\gamma A)'$ may differ from γA in both the triangle area and the value of γ, i.e. interface normal dependence of γ is possible with this approach.

We now rewrite the simplified master equation (3.15) in matrix form. First,

$$\mathbf{K}_{(h,k),(i,j)} = \frac{1}{12} \sum_{triangles} n_j n_k A + (\epsilon^{tri} + \epsilon^{tet})\delta_{ih}\delta_{jk}. \tag{3.25}$$

The dependence on indices i and h are implicit in the summation; only elements including both nodes i and h are included. Similarly,

$$\mathbf{F}_{(h,k)} = \sum_{triangles} n_k M A \sum_{r=1}^{3} \frac{\partial E}{\partial x_r^{(h)}} n_r + \epsilon^{tri} M^{tri} \frac{\partial Q^{tri}}{\partial x_k^{(h)}} + \epsilon^{tet} M^{tet} \frac{\partial Q^{tet}}{\partial x_k^{(h)}}. \tag{3.26}$$

Solving the master equation leads to a description of the continuous motion of the grain boundary network. However, topological changes such as grain collapse cannot be modelled directly. Additionally, the maximum edge length and other properties of the mesh are not constrained by the equation of motion, but can only be adjusted by retriangulation. For these reasons it is necessary to periodically halt the simulation to perform such operations. For this we use the Los Alamos Grid Toolbox (LaGrit) software package, with parameters identical to those used by Kuprat [116].

Values used for the constants in equation 3.15 are $\epsilon^{tri} M^{tri} = 10^{-5}$ and $\epsilon^{tet} M^{tet} = 10^{-15}$, with both triangle and tetrahedron quality forces equal to their aspect ratio.

3.3 Simulation method verification

To test the accuracy of each method we performed 2D and 3D simulations of an isolated circular or spherical grain shrinking with misorientation dependent boundary properties. Such a grain shrinks with a self similar shape, and the motion of the boundary with radius r and curvature κ satisfies

$$\frac{\partial r}{\partial t} = -M\gamma\kappa = -\frac{M\gamma}{r}. \tag{3.27}$$

For 2D simulations, this relation implies that the grain area A as a function of time is

$$A(t) = A_0 - 4\pi M\gamma t. \tag{3.28}$$

Simulation	Energy	Mobility	$M\gamma$ (measured)	Error (%)
MC 2D	1.00	1.00	0.994	-0.600
MC 2D	0.75	1.00	0.789	5.20
MC 2D	0.50	1.00	0.524	4.80
MC 2D	0.25	1.00	0.270	8.00
MC 2D	1.00	0.75	0.761	1.47
MC 2D	1.00	0.50	0.534	6.80
MC 2D	1.00	0.25	0.263	5.2
MC 3D	1.00	1.00	0.997	-0.300
MC 3D	0.75	1.00	0.752	0.266
MC 3D	0.50	1.00	0.502	0.400
MC 3D	0.25	1.00	0.249	-0.400
MC 3D	1.00	0.75	0.753	0.400
MC 3D	1.00	0.50	0.499	-0.200
MC 3D	1.00	0.25	0.249	-0.400

Table 3.1: Results from test cases with isolated circular grains and misorientation dependent anisotropy. Area rate of change for Monte Carlo simulations shown after conversion to "real" time.

Similarly, for 3D simulations the grain volume V follows

$$V(t) = \frac{4\pi}{3}\left[\left(\frac{3V_0}{4\pi}\right)^{\frac{2}{3}} - 4M\gamma t\right]^{\frac{3}{2}}. \tag{3.29}$$

Thus a simple means of measuring the method's ability to accurately produce anisotropic boundary motion is to compute the total area or volume of the grain at a sequence of time intervals, perform a fit using M and γ as free parameters, and compare these values to the input mobility and energy.

The results of test simulations for the Monte Carlo method are given in table 3.1. The initial grain diameter is 100 Δx, and all other simulation parameters are chosen as described above. Because the Monte Carlo method is stochastic, there are typically large deviations from the average area rate of change measured after a large number of simulations, as shown in figure 3.1. We therefore take an average over 25 simulations. Also, because it is not possible to know *a priori* the correspondence between a Monte Carlo step and "real time," we must determine an appropriate scale factor. In these simulations, where $kT = 0.7$, we find a scale factor of 2.15 MCS in 2D or 0.301 MCS in 3D by fitting the area rate of change from simulations with isotropic growth. The error in the area or volume rate of change from each Monte Carlo simulation are similar, but they do not appear to be systematic.

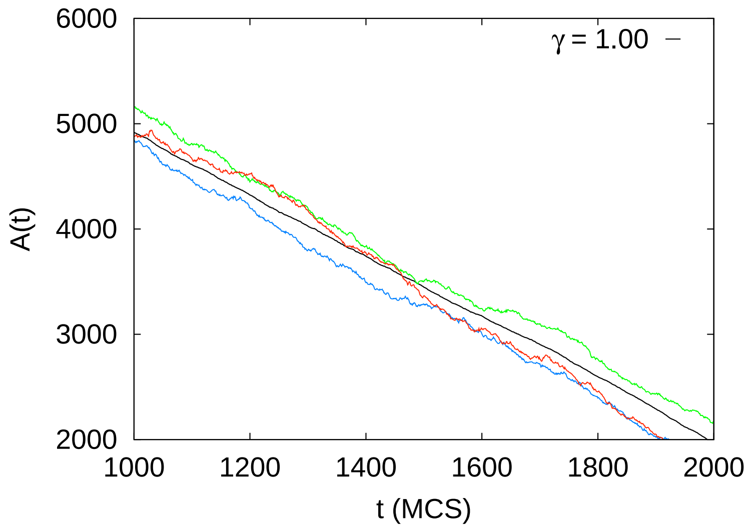


Figure 3.1: Area rate of change of circular shrinking grains for Monte Carlo with isotropic boundary properties: average area rate of change and results from three individual simulations.

Similar tests were performed with spherical shrinking grains using the moving finite element method by Kuprat [116]. The error in $\partial A/\partial t$ was found to scale with $(\Delta\theta)^2$, where θ is the angular resolution of the meshed sphere.

The mesh size of the moving finite element mesh is important in determining the physical behavior of the simulation. In particular, poor mesh quality may prevent local equilibrium at triple junctions. We have computed the distribution of dihedral angles within the MFE mesh during a simulation with isotropic interfacial properties. Figure 3.2 shows this distribution after nearly one quarter and one half of the grains have been eliminated, respectively. The average dihedral angle is near the ideal values of 120°, while the standard deviation are found to be 8.5° and 5.7°. This suggests that as the mesh evolves, the mesh better captures triple line equilibrium. This is presumably due to the fact that grains are eliminated and the average number of facets per grain increases.

3.4 Microstructure generation

Monte Carlo. Because we cannot expect grain boundaries with relatively few pixels or voxels to accurately reproduce curvature driven motion, we begin each simulation with a relatively coarse

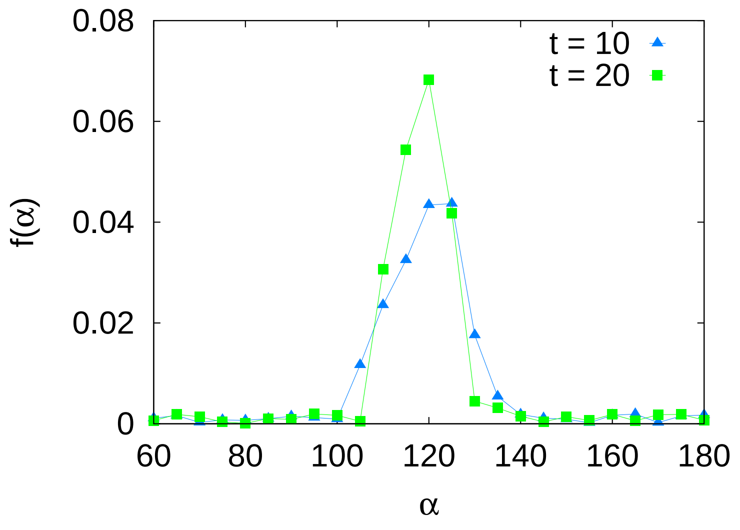
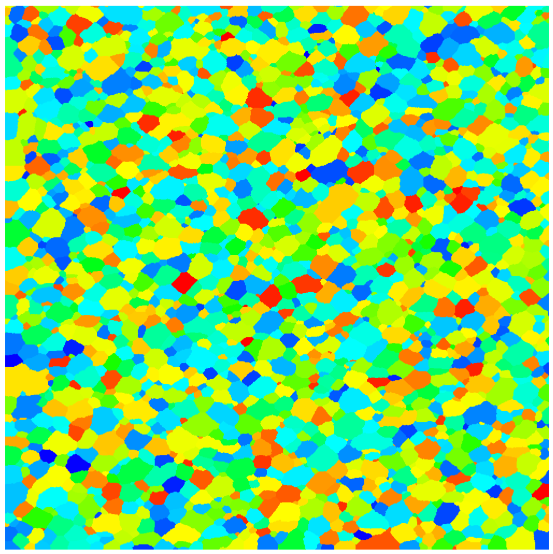


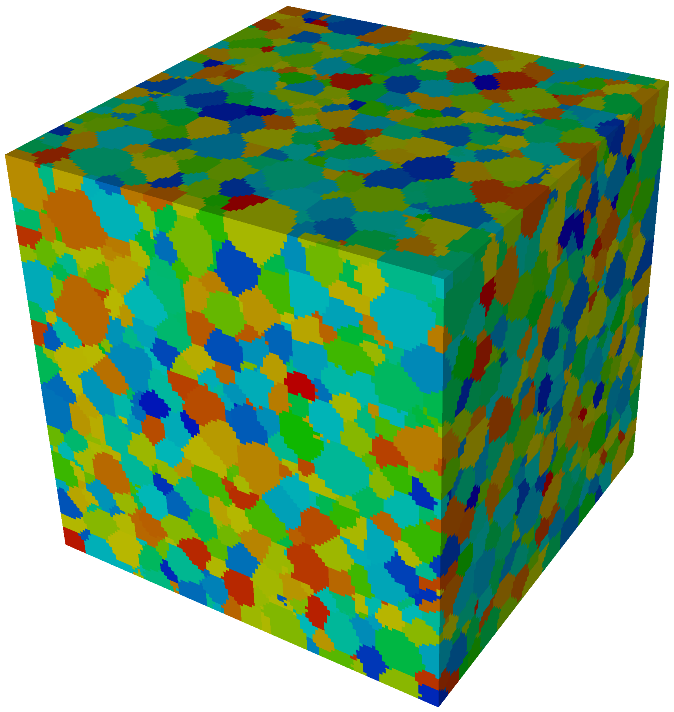
Figure 3.2: Frequency of dihedral angles in MFE simulations as a function of time.

microstructure. For Monte Carlo simulations, we use either a $4,096^2$ (2D) or 256^3 (3D) site lattice. To produce the initial microstructure, unique spin numbers were assigned to each lattice point on a Monte Carlo grid, and the microstructure was coarsened with isotropic interfacial properties until 68,651 (2D) or 35,751 (3D) grains remained. Portions of each initial microstructure are shown in figure 3.3. Relative to the lattice, the grains and grain boundaries were reasonably well resolved, with an average of approximately 244 (2D) or 470 (3D) lattice sites per grain. The initial 2D Monte Carlo microstructure had 203,271 grain boundaries, while the 3D microstructure had 238,412.

Moving finite elements. Our initial geometry for moving finite element simulations was generated as follows. The unit cube was partitioned into a regular grid of 50^3 cells. A node was placed at the order of each cell. The nodes were connected to create a regular tetrahedral mesh. This resulted in a mesh of 750,000 tetrahedra of equal volume. Grain identifiers were randomly assigned to 5,000 of the tetrahedra while following the condition that no two grain centers lie in adjacent tetrahedra. This mesh was evolved isotropically until 2,578 grains remained, figure 3.4. The coarsening process included a sequence of mesh quality operations as described above.



(a) 1/16 portion of 2D Monte Carlo grid.



(b) 1/8 portion of 3D Monte Carlo grid.

Figure 3.3: Initial microstructures for 2D and 3D simulations.

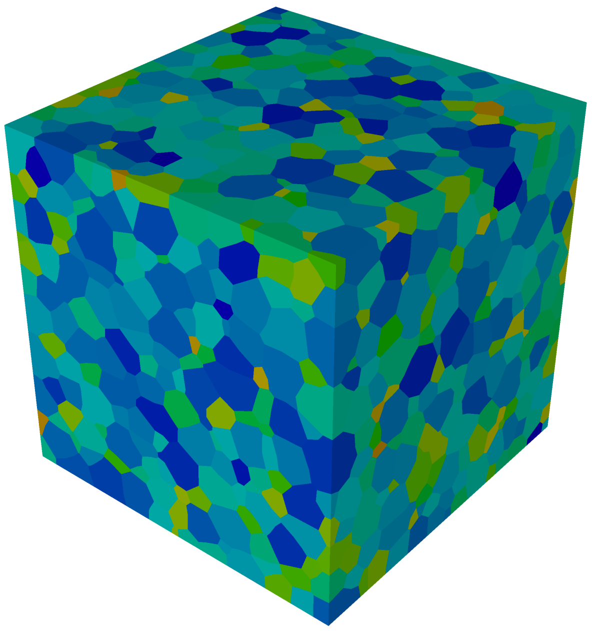


Figure 3.4: Initial microstructure for 3D FEM simulations.

Orientation assignment. Each grain identifier is mapped to a unique crystallographic orientation. To produce a random orientation texture, we generate (Bunge) Euler angles [42]

$$\begin{aligned} \varphi_1 &= 2\pi r_1 \\ \Phi &= \cos^{-1}(1 - 2r_3) \\ \varphi_2 &= 2\pi r_2 \end{aligned} \tag{3.30}$$

where each r_i, $1 \leq i \leq 3$, is a random floating point number in the unit interval. Other orientation textures can be produced by producing a set of random orientations, then selecting a subset using some predetermined rule. We will describe the relevant selection rules for simulations with nonrandom initial orientation textures as they appear.

We typically perform simulations where the time evolution of any grain boundary depends on both its local geometry and the particular form of its energy and mobility functions. By performing simulations with different grain identifier to orientation mappings, we can effectively sample many sets of grain boundaries without generating more than one microstructure. For each set of interfacial properties we use several such mappings.

3.5 Microstructure analysis

Computating the area (2D) or volume (3D) of a grain on a Monte Carlo grid simply consists of counting all sites with some given grain identifier. We make the assumption that each grain constitutes one connected component, which is sometimes not the case in Monte Carlo simulations, although this is due primarily to random noise at interfaces. For an unstructured mesh, a similar procedure requires computation of the area or volume of individual elements.

Measuring geometric features of a microstructure that depend on crystallographic orientation requires a choice in what constitutes a single grain or grain boundary. In experimental measurements, the limited accuracy of orientation measurements imposes a lower bound for grain boundary misorientation. This is not the case with simulations. Since each volume element has a discrete grain identifier which is associated with a fixed orientation, there is no misorientation resolution limit. In our simulations where interfacial energy approaches zero as misorientation approaches zero, we occasionally observe two or more grains of low misorientation ($< 1^\circ$) meeting together and then evolving as essentially a single grain. We have chosen to impose a minimum misorientation angle of 2° in all or our analysis so that our simulations are more directly comparable to experiment.

A suitable definition of grain boundary area on a Cartesian grid is required for the measurement of area weighted texture measures in Monte Carlo simulations. Given a particular grid site, we count each pair of neighboring sites, either 8 (2D) or 26 (3D), with unlike grain identifiers as a unit grain boundary area. It is possible that this approach produces different results with a different choice of neighborhood. Likewise, we might also consider weighting the contribution of each neighbor by some factor other than one. While other neighbor counting or weighting schemes certainly compute different absolute areas for a given boundary, we will show that the relative area measurements are reasonably insensitive to the details of the method. The boundary area measurement in an unstructured mesh is obviously more straightforward, as triangle areas can be computed directly.

Computing number weighted texture measures involves the same procedure used in the area measurement, but individual grain boundaries are given unit weight. To simplify this calculation, we assume that if two grains share a boundary, they share a single boundary. Under this assumption, the data collected in an area measurement provides all the necessary information for a number measurement. For example, in a Cartesian grid, the set of all neighbor pairs with unlike grain identifiers contribute one grain boundary to the number count, while contributing a value equal to the total number of such pairs to the area measurement.

Chapter 4

Misorientation dependent anisotropy

One of the objectives of this work is to determine a quantitative relationship between grain boundary energy and mobility and the interface texture resulting from grain growth. In full generality, this seems to be an overwhelming task — each boundary property, as well as the distribution of boundary types, is a function of five parameters. However, it is certainly possible to study energy and mobility functions with a reduced number of independent parameters, and in fact previous work has demonstrated that even disorientation angle based properties lead to interface texture development [25, 28, 30–32, 34].

In this chapter and the next, we examine misorientation and inclination dependent anisotropy separately. Here, we limit misorientation to a single parameter (disorientation angle). In the limiting case that interface texture can be written as a function of boundary properties alone, then the results obtained here should apply to systems with full misorientation dependent anisotropy. The results presented here are from grain growth simulations using Monte Carlo and phase field methods as explained in the previous chapter.

Misorientation texture can develop as a result of changes in either the relative number of grain boundaries, the average area of grain boundaries, or both. Experimental evidence suggests that both occur [5]. Holm et al. introduced a model for interface texture development that explains the changes in area weighted MDFs by boundary lengthening, but does not predict changes in number weighted MDFs [28]. In other computational studies, the problem of how number weighted MDFs develop has largely been ignored. In this chapter we demonstrate that number weighted MDFs are measurably affected by anisotropic interfacial properties. These results provide the motivation for a new model for predicting changes in the number and area weighted MDFs that occur during grain growth.

4.1 Simulations

While a general misorientation is characterized by three independent parameters, the grain boundary properties used in this work are explicit functions of a single parameter, the grain boundary disorientation angle. Note that such functions incorporate crystallographic symmetry implicitly in the computation of disorientation angles. Grain boundary energy and mobility will always take one of three functional forms, which are used in both 2D and 3D simulations (orientations are always three dimensional). First, the isotropic function

$$\gamma_{\mathrm{iso}}(\theta) = 1 \tag{4.1}$$

which is unity for all disorientation angles. Second, the Read-Shockley function,

$$\gamma_{\mathrm{RS}}(\theta) = \begin{cases} \frac{\theta}{\theta'}\,[1 - \ln(\,\frac{\theta}{\theta'}\,)] & \theta \le \theta' \\ 1 & \theta > \theta' \end{cases} \tag{4.2}$$

where θ is the disorientation angle and θ' is the high angle grain boundary cut-off. Lastly, a step function

$$\gamma_{\mathrm{step}}(\theta) = \begin{cases} \frac{2}{5} & \theta \le \theta' \\ 1 & \theta > \theta'. \end{cases} \tag{4.3}$$

Although we used γ in the definitions above, mobility functions M_{iso}, M_{RS}, and M_{step} are defined in a similar way.

Monte Carlo simulations were performed with a number of different energy and mobility func-

Energy	Mobility	Symmetry	β (2D)	β (3D)	Boundary Cond.
$\gamma_{\text{iso}}(\theta)$	$M_{\text{iso}}(\theta)$	O	0.7	1.3,1.4,1.5	F,P
$\gamma_{\text{iso}}(\theta)$	$M_{\text{RS}}(\theta)$, $\theta' = 45°$	O	0.7	1.5	F
$\gamma_{\text{RS}}(\theta)$, $\theta' = 15°$	$M_{\text{iso}}(\theta)$	O	0.7	1.5	F
$\gamma_{\text{RS}}(\theta)$, $\theta' = 30°$	$M_{\text{iso}}(\theta)$	O	0.7	1.5	F
$\gamma_{\text{RS}}(\theta)$, $\theta' = 45°$	$M_{\text{iso}}(\theta)$	O	0.7	1.3,1.4,1.5	F,P
$\gamma_{\text{RS}(\theta)}$, $\theta' = 45°$	$M_{\text{iso}}(\theta)$	D_6	0.7	1.5	F
$\gamma_{\text{step}}(\theta)$, $\theta' = 30°$	$M_{\text{iso}}(\theta)$	O	0.7	1.5	F

Table 4.1: Summary of Monte Carlo simulation parameters. All simulations listed were performed with 2D and 3D microstructures. F and P denote finite and periodic boundary conditions, respectively.

tion combinations, as well as several different choices for the lattice temperature β and boundary conditions. A summary of the simulation parameters is given in table 4.1.

Additionally, several simulations were performed with anisotropic energy and non-random initial orientation distributions. In these cases, orientations were generated randomly as described in the previous chapter, but were selected with a probability based on their proximity to a single favored orientation coinciding with the sample coordinate system. Using the crystal disorientation from sample coordinates θ_S, we chose

$$P(\theta_S) = e^{-\alpha\theta_S} \tag{4.4}$$

where $\alpha = 4.20, 6.05, 9.45,$ or 16.80. Larger values of α produce orientation textures that increasingly favor the preferred orientation. These simulations were all performed using the 3D microstructure with γ_{RS}, $\theta' = 45°$, isotropic mobility, $kT = 1.5$, and finite boundary conditions.

4.2 Results

4.2.1 General observations

The time-dependent behavior of grain size in simulations with anisotropic interfacial properties is approximately the same as that in isotropic growth, as shown in figure 4.1. The kinetic exponent n for isotropic growth in 3D is found to be $n = 1.49$, which is near the theoretical volume rate of change exponent value of 1.5 [123]. This is nearly identical to the exponent for growth with anisotropic energy in all cases. The kinetic exponent for average grain area versus time for isotropic growth in

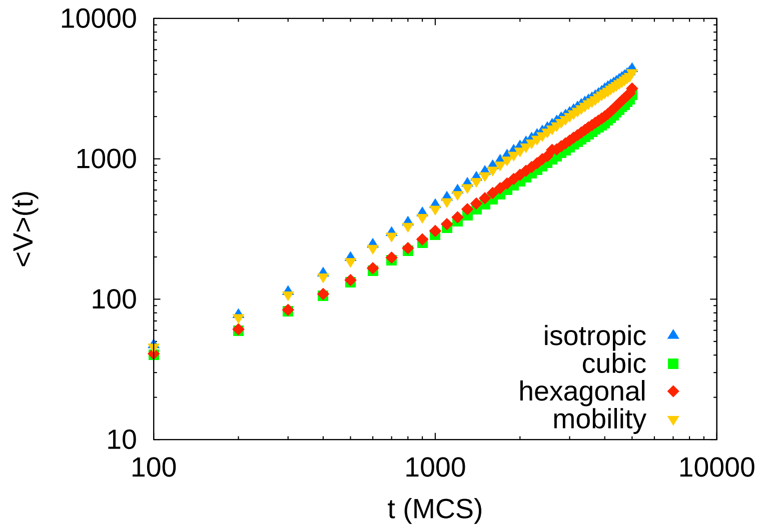
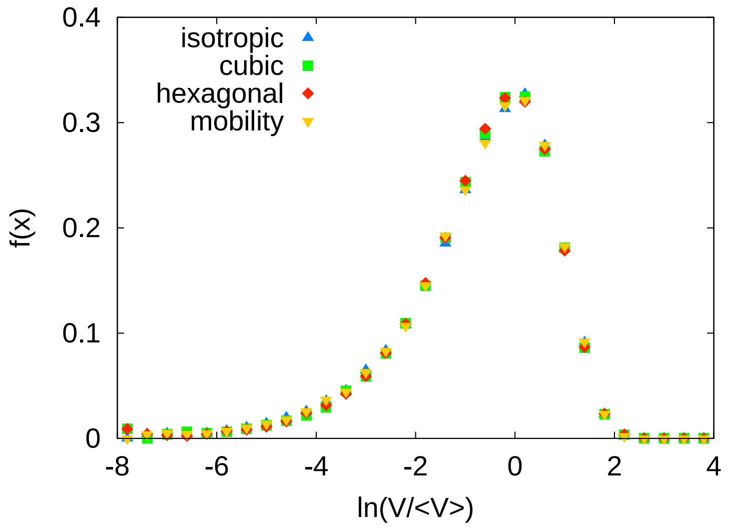


Figure 4.1: Grain volume V as a function of time t for 3D Monte Carlo simulations. Here, "cubic" and "hexagonal" imply $\gamma_{\mathrm{RS}}(\theta)$ with $\theta' = 45°$ and isotropic mobility, while "mobility" implies $M_{\mathrm{RS}}(\theta)$ with $\theta' = 45°$ and isotropic energy.

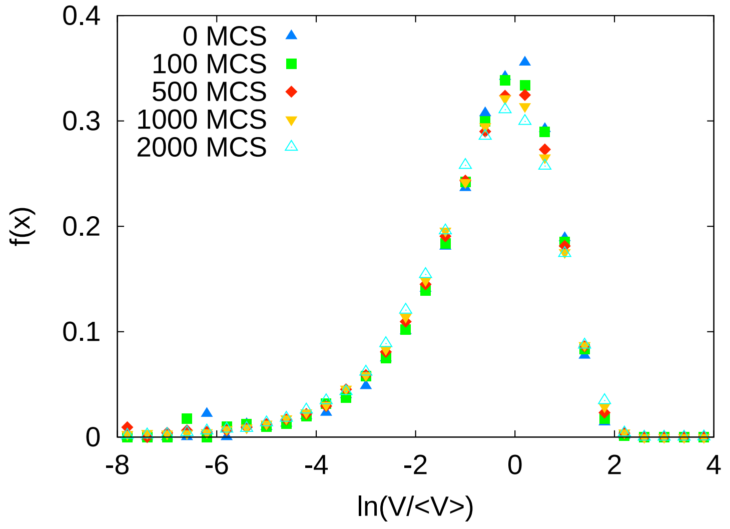
2D is found to be $n = 0.98$, again near the theoretical area rate of change value of 1.0. It is clear that the absolute rate of grain growth for simulations with anisotropy is slower than with isotropic properties. Presumably, the rate is slower with anisotropy because the average energy or mobility of boundaries in such cases is less than one. We then expect that as we increase the fraction of disorientation space with energy or mobility less than one, the grain growth rate decreases, which is satisfied here.

The grain size distribution in all simulations appears to be identical to that in the isotropic case, figure 4.2. In our computations we have excluded grains less than about 10 pixels/voxels. We also find that the grain size distribution does not change appreciably with time.

The grain morphology during 2D and 3D growth with anisotropic properties is illustrated in figure 4.3. Grain shapes appear to be mostly equiaxed, and we can visually locate several triple junctions with dihedral angles which are greater or less than the isotropic equilibrium value of 120°. The average topological properties of grains are roughly the same with or without anisotropy. Excluding grains on the domain boundary, we find that grains in 3D have an average of about 13.8 faces and 5.3 edges, while grains in 2D have 6.0 faces. These values remain approximately constant



(a) Grain size distribution at 500 MCS.



(b) Grain size distribution for simulation with $\gamma_{\mathrm{RS}}, \theta' = 45°$.

Figure 4.2: Grain size distribution for various 3D Monte Carlo simulations.

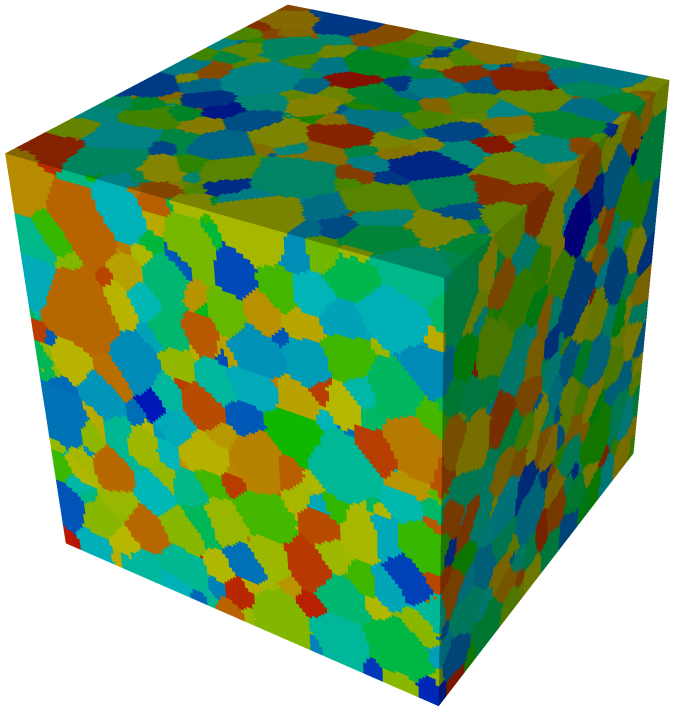


Figure 4.3: Microstructure from grain growth with γ_{RS}, $\theta' = 45°$, 2000 MCS. 1/8 of the simulation domain is shown.

through time, and are similar to values reported elsewhere [124].

4.2.2 Time dependence

In every simulation performed, we find that after an initial transient state, both area and number weighted MDFs reach what appear to be steady-state distributions. To quantify the transient, we introduce a value $\Delta(t)$, defined as

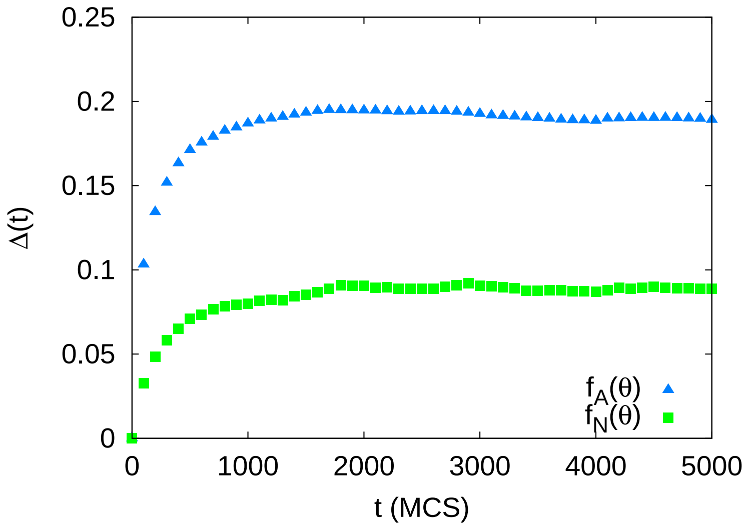
$$\Delta(t) = \int |f(\theta, t) - f(\theta, 0)| \, d\theta \tag{4.5}$$

where $f(\theta, t)$ represents either an area or number weighted MDF depending on the context. Thus $\Delta(t)$ is just the L1 norm of the difference between a pair of distributions, one of which is always the initial (random) MDF. Note that $\Delta(t)$ approaching a constant as t increases is a necessary condition for the function $f(\theta, t)$ to reach a steady state.

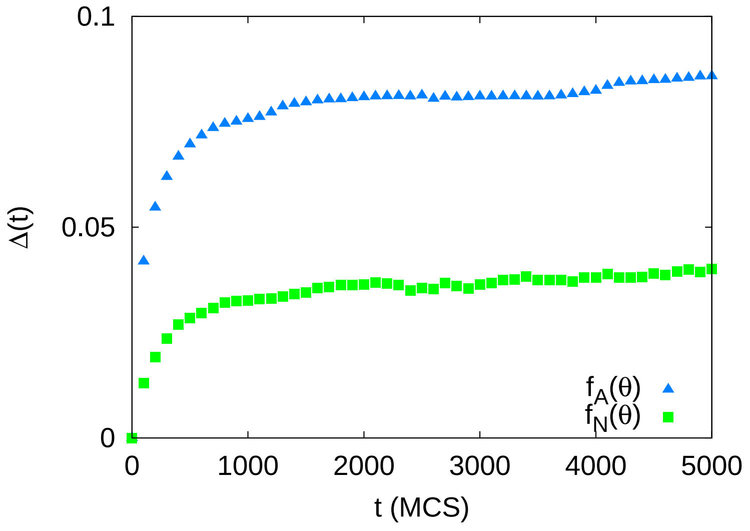
In figure 4.4 we show the time dependence of $\Delta(t)$ for the area and number weighted MDF during a particular simulation. It is clear that for the simulation with cubic crystal symmetry, $\Delta(t)$ approaches some constant value. In other simulations, at long times $\Delta(t)$ continues to change slowly, although this change is on the order of $\Delta(t)$ for isotropy. This implies that further changes in the MDFs are due primarily to increasing measurement inaccuracy, as the total number of boundary observations decreases rapidly while time progresses. We observe that the transient regime is consistently longer for simulations with larger energy anisotropy. Grain growth occurs more slowly in the 2D simulations than in 3D (see figure 4.1), and subsequently the transient times for 2D are longer.

4.2.3 Property dependence

We observe a measurable change in the number and area weighted MDFs produced during grain growth with anisotropic interfacial energy. This is true in both two and three dimensions and for each energy function used. Figures 4.5 and 4.6 show the steady state number and area weighted MDFs for the simulations with cubic or hexagonal crystal symmetry and Read-Shockley energy functions with 45° cutoff angle after a significant portion of the original grains have been eliminated by grain growth. In each case, the lowest energy boundaries are those closest to the origin. These

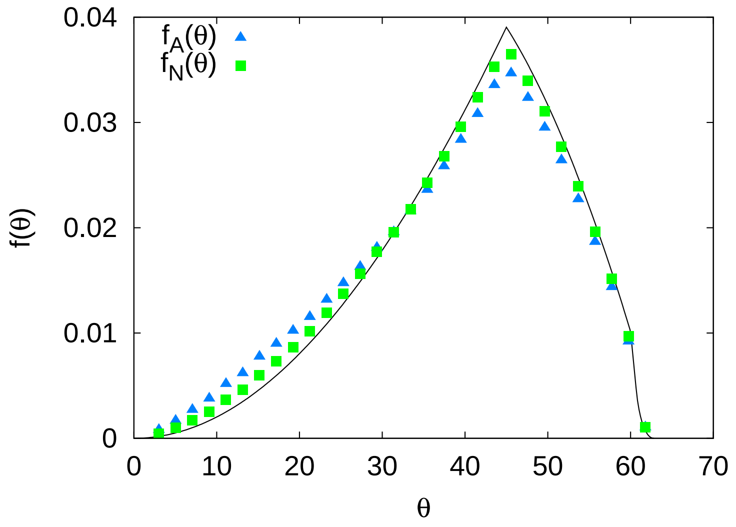


(a) Cubic crystal symmetry.

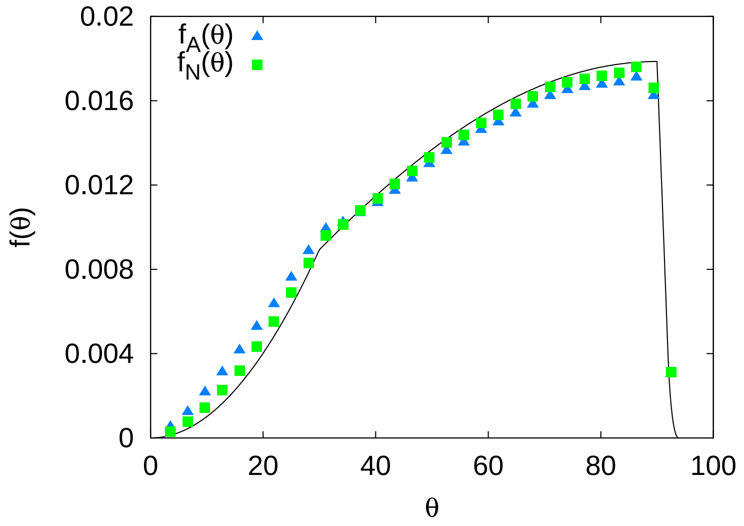


(b) Hexagonal crystal symmetry.

Figure 4.4: Plots of $\Delta(t)$ for select 3D Monte Carlo simulations with energy functions $\gamma_{\mathrm{RS}}(\theta)$, $\theta' = 45°$ and isotropic mobility.

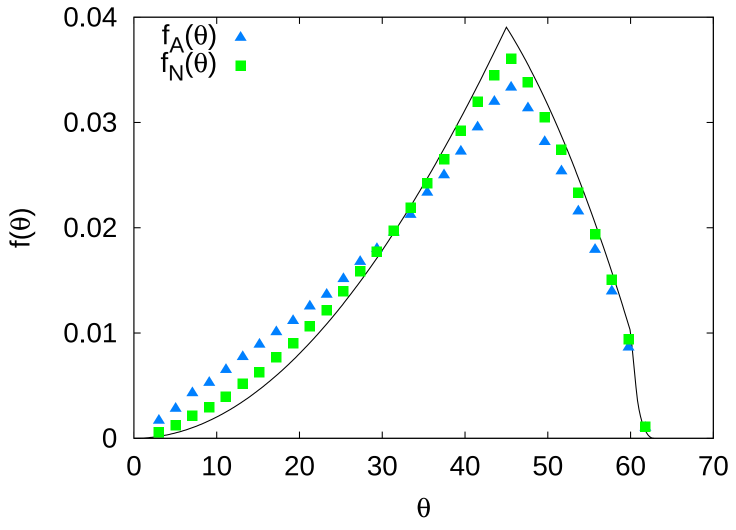


(a) Cubic crystal symmetry.

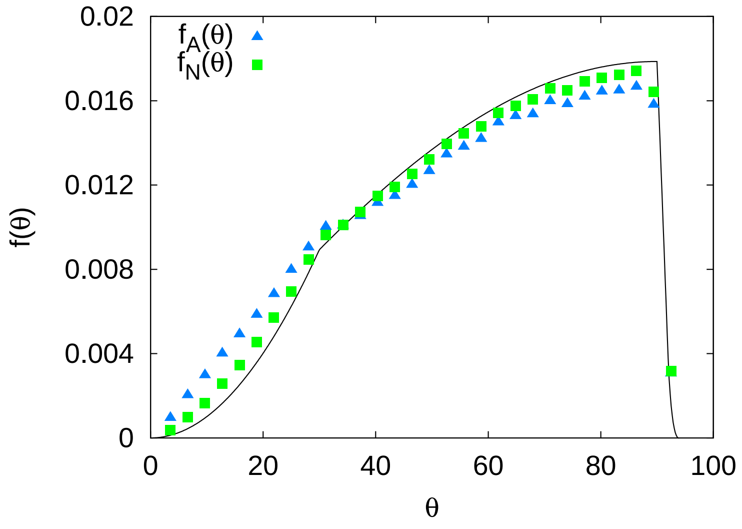


(b) Hexagonal crystal symmetry.

Figure 4.5: Misorientation distribution functions after 2D grain growth with energy functions $\gamma_{\mathrm{RS}}(\theta)$, $\theta' = 45°$ and isotropic mobility at 1000 MCS. In each case, the average grain area has doubled.



(a) Cubic crystal symmetry.



(b) Hexagonal crystal symmetry.

Figure 4.6: Misorientation distribution functions after 3D grain growth with energy functions $\gamma_{\mathrm{RS}}(\theta)$, $\theta' = 45°$, isotropic mobility, at 500 MCS. In each case, the average grain volume has doubled.

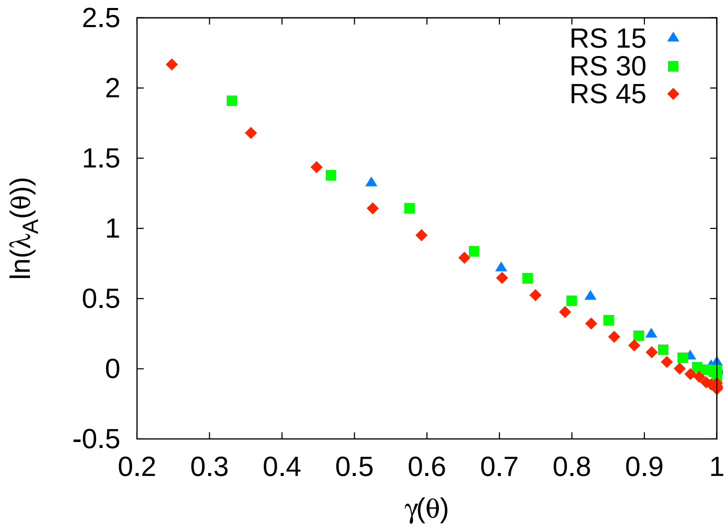
boundaries have increased in both number and average area relative to the higher energy boundaries at larger disorientation angles. The area weighted MDF in all cases shows greater anisotropy than the number weighted MDF, which is expected since the relative area of a boundary type is just its relative number multiplied by its average area. In multiples random, the area and number weighted MDFs appear to have an approximately linear inverse relationship with grain boundary energy, as shown in figure 4.7. We note that the slope of the best fit line through the area weighted data is almost identically twice the best fit slope through the number weighted data.

There appears to be a consistent inverse proportionality between the grain boundary energy and the average boundary area, as shown in figure 4.8. In fact, the average area appears to be an approximately linear function of the grain boundary energy for most of the disorientation angle space. Figure 4.8 also demonstrates that the relationship between grain boundary energy and average area may not always be one-to-one. This implies that the grain boundary's disorientation has some influence on the average boundary area. However, the step function energy used here is non-physical, and it appears that this relationship is satisfied well by the more realistic continuous energy functions.

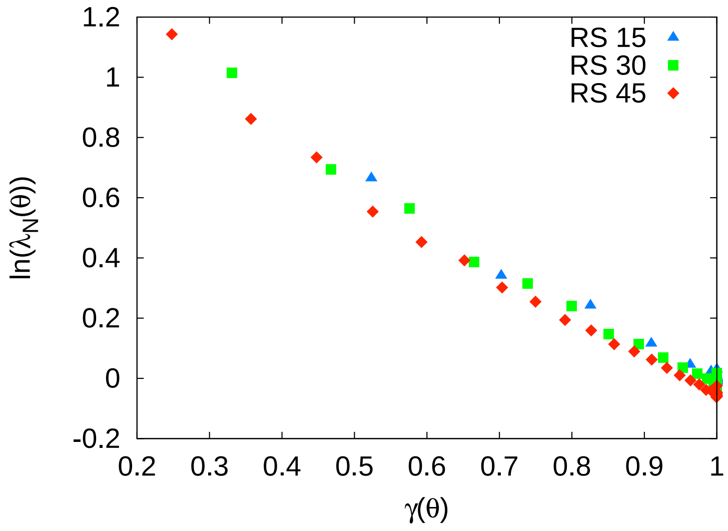
The effect of increasing energy anisotropy on the area and number weighted MDFs is illustrated in figure 4.9. We find that increasing energy anisotropy leads to both area and number weighted MDFs which are increasingly different in comparison to those produced by isotropic grain growth. The effect of mobility anisotropy on the MDF is noticeably weaker than that of energy anisotropy, as shown in figure 4.10. In fact, both area and number weighted MDFs resulting from grain growth with mobility anisotropy are negligibly different from those developed with isotropic properties. These results are in agreement with previous findings using phase field and moving finite element methods [31, 32, 37].

4.2.4 Orientation texture dependence

The area and number weighted MDFs presented above for simulations with anisotropic energy all show some slight deviation from the distribution for a collection of random boundaries. Clearly, the orientation texture of the microstructure has a large effect in determining how the MDFs might evolve. We now describe results from simulations with non-random orientation texture.

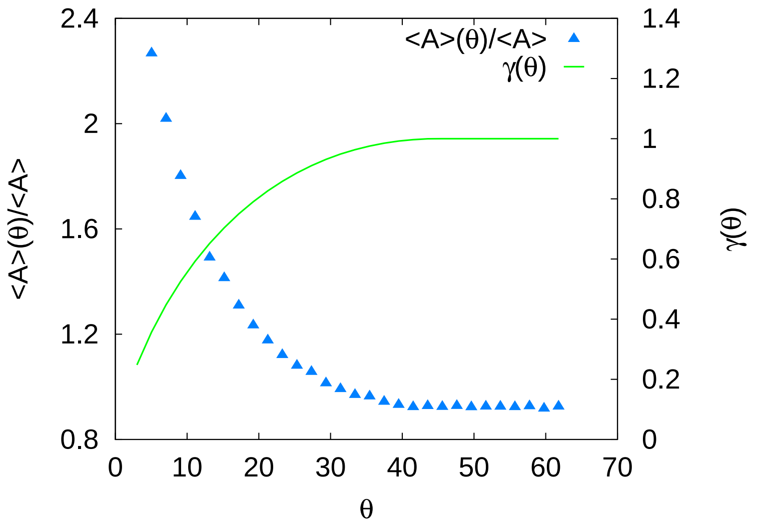


(a) Area weighted MDF in multiples random.

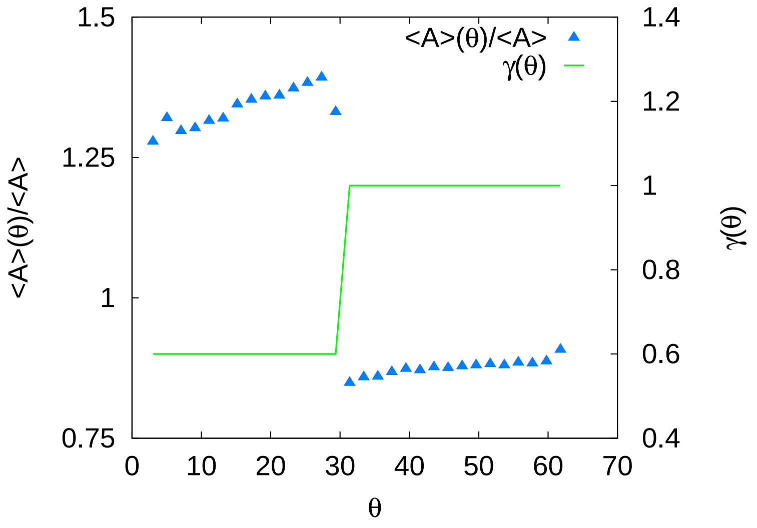


(b) Number weighted MDF in multiples random.

Figure 4.7: Grain boundary population in multiples random λ as a function of energy γ in several 3D simulations, 500 MCS.

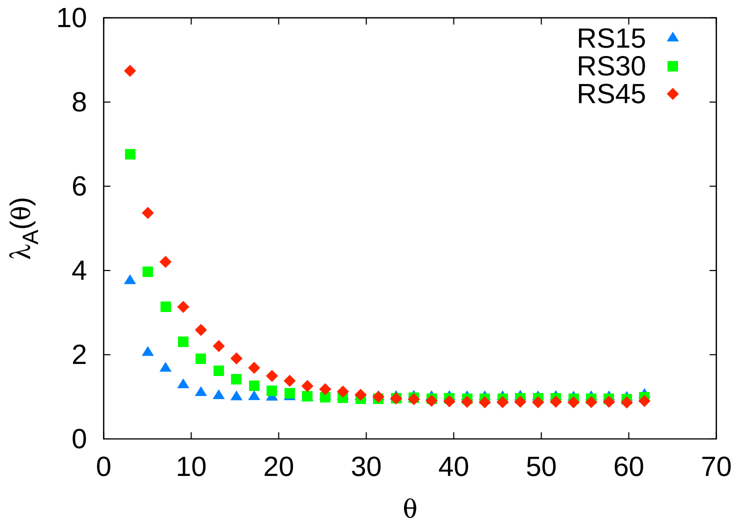


(a) $\gamma_{\mathrm{RS}}(\theta)$, $\theta' = 45°$.

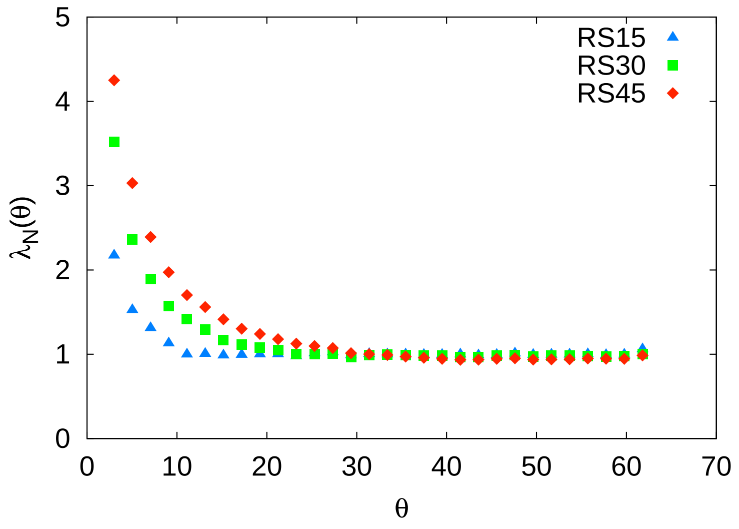


(b) $\gamma_{\mathrm{step}}(\theta)$, $\theta' = 30°$.

Figure 4.8: Normalized average boundary area $\langle A\rangle(\theta)/\langle A\rangle$ and grain boundary energy γ as a function of disorientation θ for Monte Carlo simulations with anisotropic energy, isotropic mobility, and cubic crystal symmetry at 500 MCS.

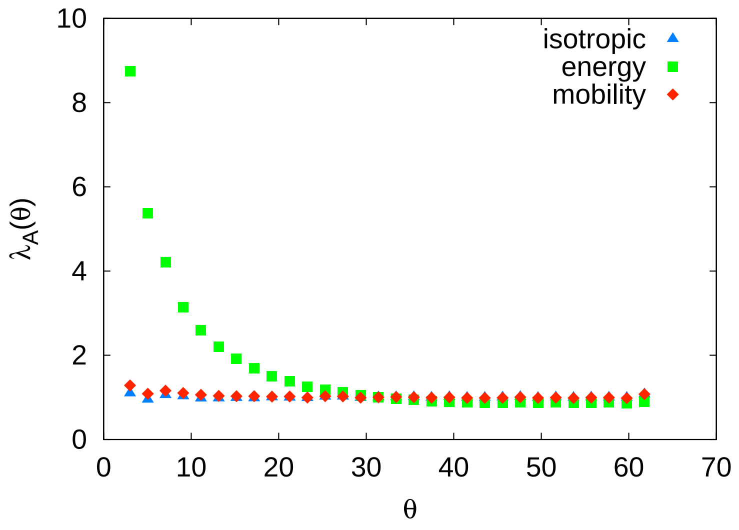


(a) Area weighted MDFs in multiples of random distribution (MRD).

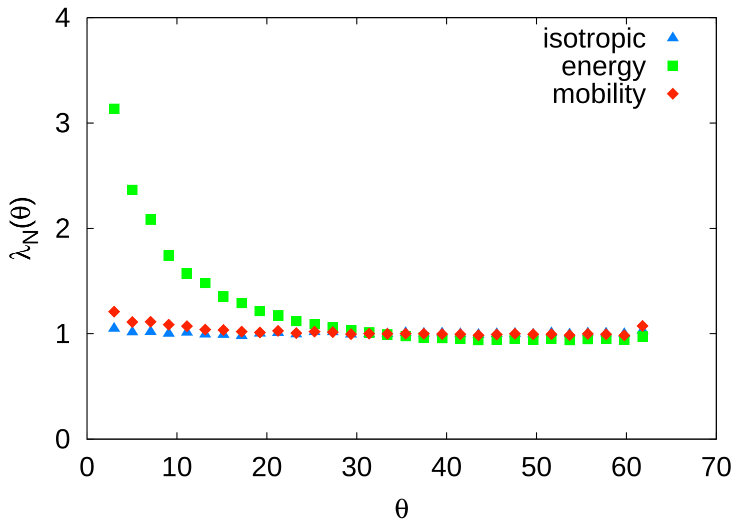


(b) Number weighted MDFs in multiples of random distribution (MRD).

Figure 4.9: Area and number weighted MDFs in multiples random for various Read-Shockley type energy functions and isotropic mobility. Data from 3D Monte Carlo simulations at 500 MCS.



(a) Area weighted MDFs in multiples of random distribution (MRD).



(b) Number weighted MDFs in multiples of random distribution (MRD).

Figure 4.10: Area and number weighted MDFs for Read-Shockley type energy or mobility functions. Data from 3D Monte Carlo simulations at 500 MCS.

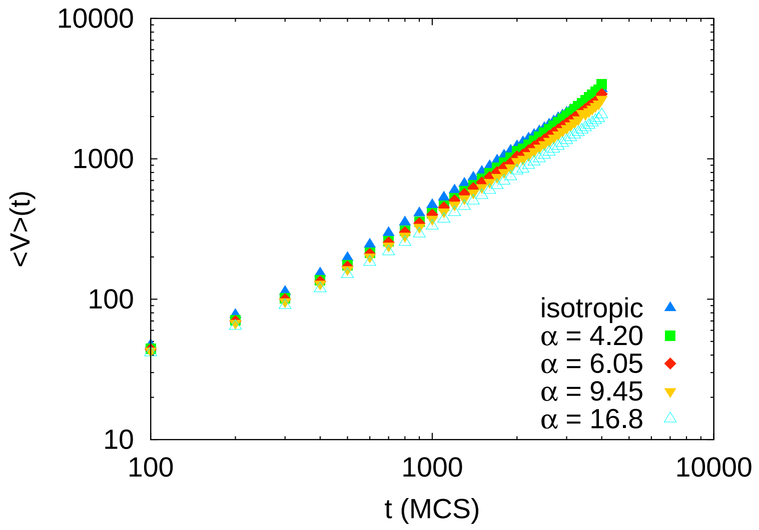


Figure 4.11: Average grain volume $\langle V \rangle$ as a function of time for simulations with non-random orientation texture.

First, we find that the kinetics of such simulations are nearly identical to the cases with random orientation texture. Figure 4.11 shows the average grain volume as a function of simulation time. As above, the grain growth exponent in simulations with anisotropic energy tends to be slightly lower than with isotropy. Likewise, the grain size distribution appears to be similar for all simulation conditions as well as constant through time, figure 4.12.

The average area of grain boundaries in simulations with non-random orientation texture follows the same behavior as with random orientation texture, figure 4.13. These functions appear to be identical, but each shifted along the y-axis, likely due to the fact that those simulations with greater orientation texture have far fewer high angle boundaries and therefore larger values of $\langle A \rangle$ at earlier simulation times.

Despite these similarities, we find qualitatively different behavior of the area and number weighted MDFs. In particular, there appears to be no steady state within the time interval simulated, figure 4.14. Even in the case where $\alpha = 4.20$, the initial MDFs are not quantitatively far from the Mackenzie distribution yet the resulting MDFs at later times are increasingly far from those obtained in simulations with random orientation texture. As the average boundary areas mimic those found with random orientation texture, it must be that this difference is due to a drastic increase in the

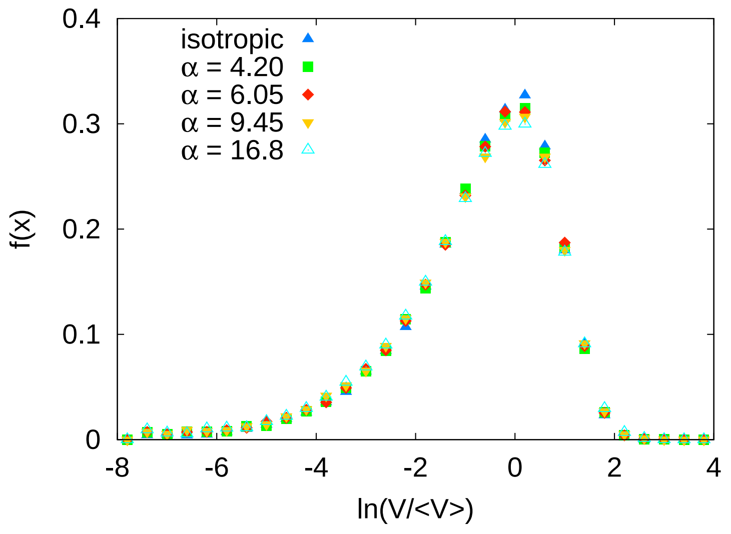


Figure 4.12: Grain size distribution for simulations with non-random orientation texture at 500 MCS.

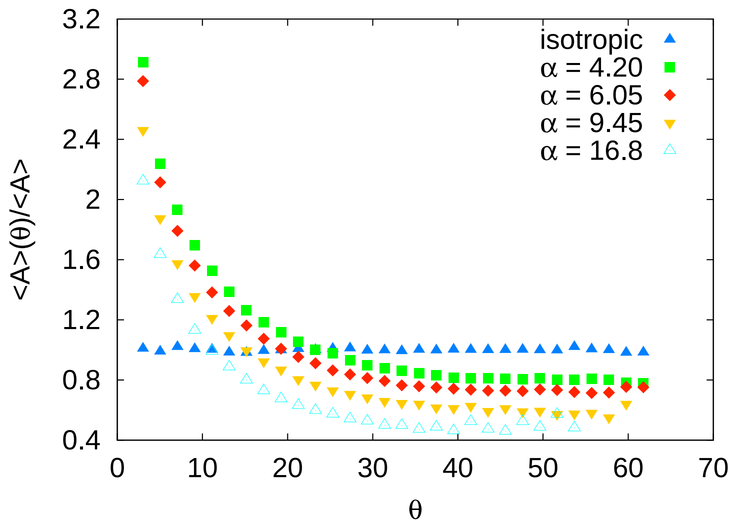
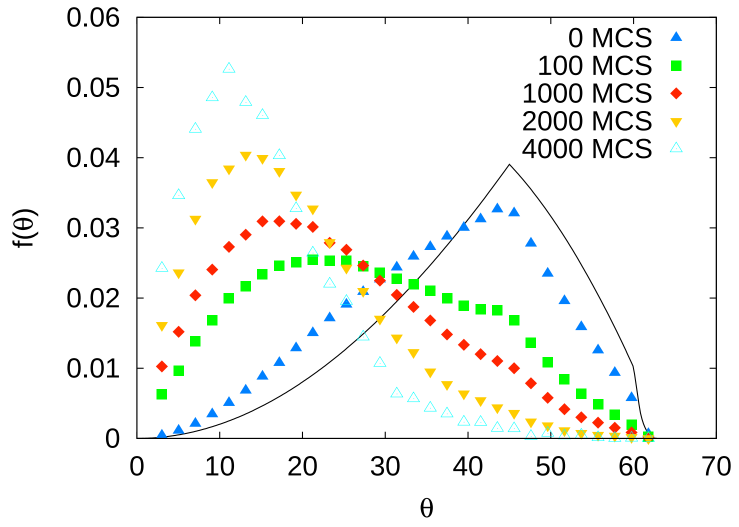
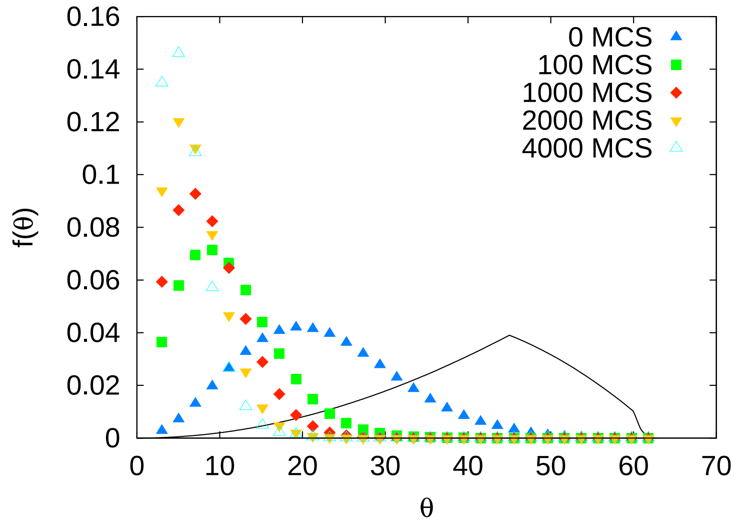


Figure 4.13: Relative average area $\langle A\rangle(\theta)/\langle A\rangle$ of grain boundaries as a function of disorientation angle θ for simulations with non-random orientation texture at 500 MCS.



(a) Simulation with $\alpha = 4.20$.



(b) Simulation with $\alpha = 16.80$.

Figure 4.14: Area weighted MDFs measured for simulations with non-random orientation texture.

number of low angle grain boundaries. We cannot exclude the possibility these MDFs eventually reach a steady state without simulating to longer times.

The mechanism for this seems to be amplified by a continuously stronger orientation texture. Figures 4.15 and 4.16 show orientation distribution functions measured at the initial state and at 4000 MCS for the simulation with $\alpha = 4.20$. The initial orientation distribution shows preferential alignment with sample axes, as expected, while the orientation distribution at 4000 MCS shows that this preferred texture has become stronger through the grain growth. Because we have chosen to use an energy function that is an increasing function of disorientation, we might expect that with our chosen orientation texture the high energy boundaries typically exist on those grains that are not aligned with the sample axes. That is, we expect a correlation between high energy grain boundaries and grains that do not have the preferred orientation. We have shown that high energy boundaries are eliminated preferentially by the grain growth process, and presumably this affects the lifetime of such grains. In simulations with random orientation texture, there should be no correlation between high energy boundaries and any grain orientation, explaining why there is no orientation texture development in such cases.

The orientation texture of the microstructure deteremines in some part what the expected MDFs should be. We define the TMDF as the probability density of grain boundary types given by random selection of grain pairs from the polycrystal orientation distribution function (ODF). For example, in systems with random orientation texture the texture weighted MDF is just the (cubic) Mackenzie distribution or its analogue for other crystal structures. The TMDF can be estimated numerically from orientation texture data by randomly selecting existing pairs of grain orientations, thereby generating a list of possible grain boundaries with which to compute the TMDF in the same way that we compute the MDF. Figure 4.17 shows the texture weighted MDF as a function of time for two simulations with non-random orientation texture. The changing orientation texture increases the bias towards low angle grain boundaries, and this appears to be the primary reason for the differences between the MDFs measured in these and the random orientation texture simulations.

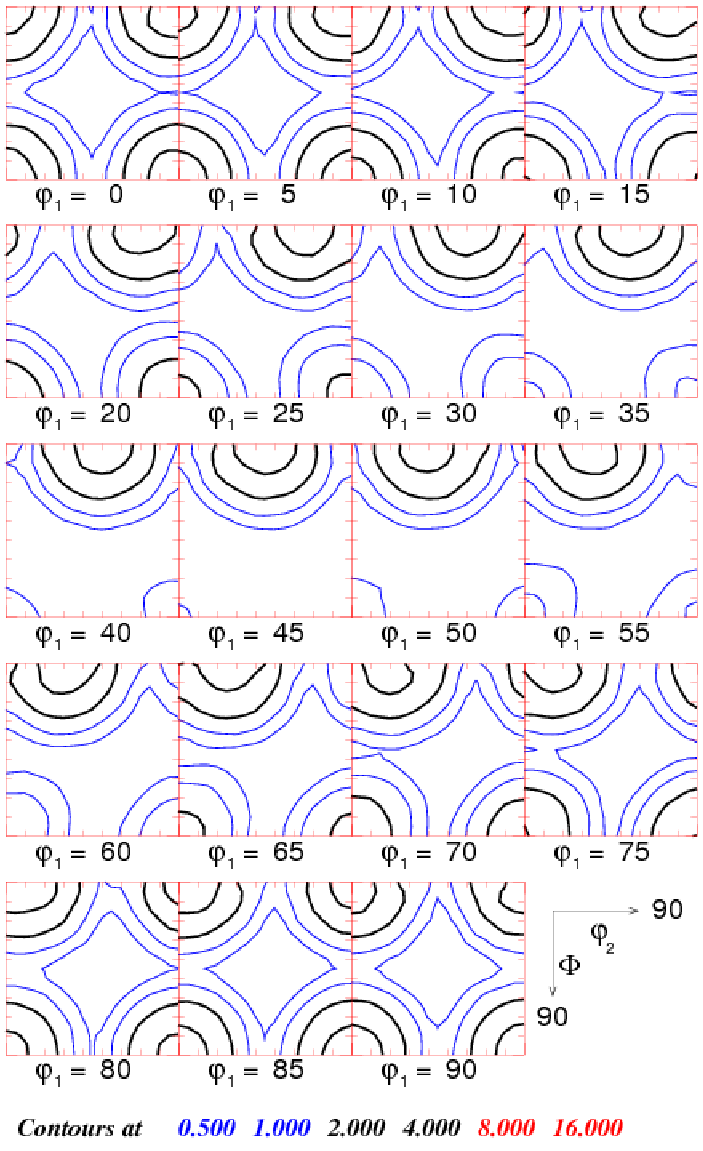


Figure 4.15: Orientation distribution function for simulation with non-random initial orientation texture with $\alpha = 4.20$ at 0 MCS.

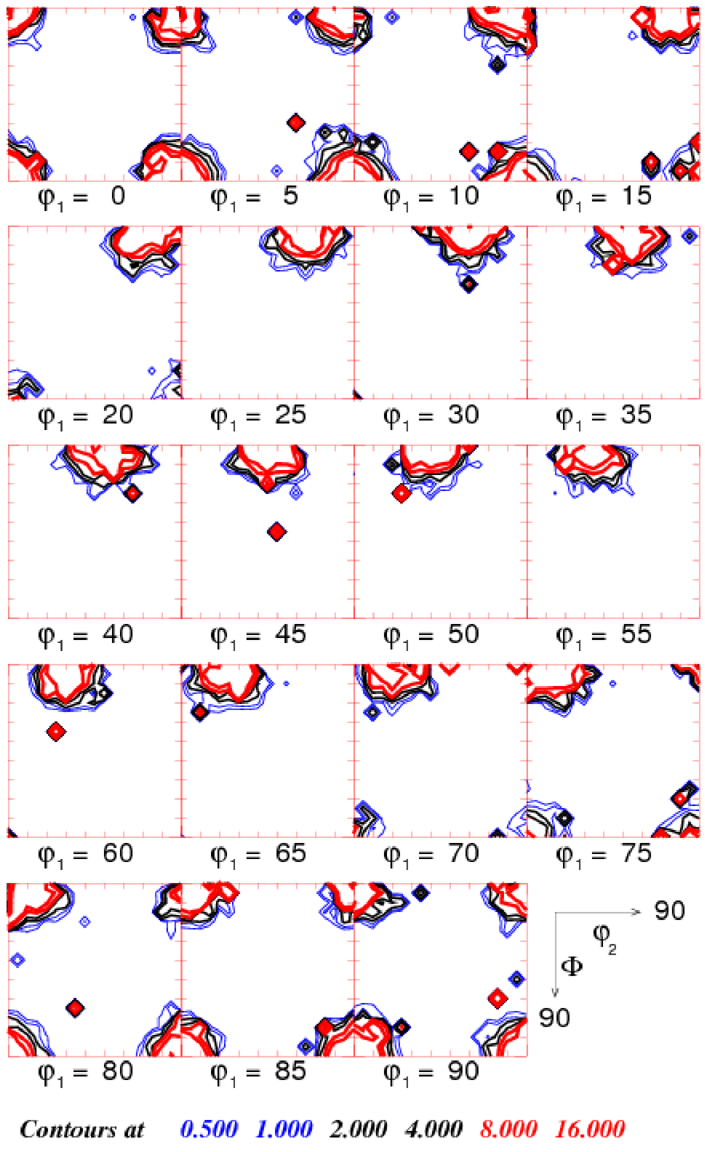
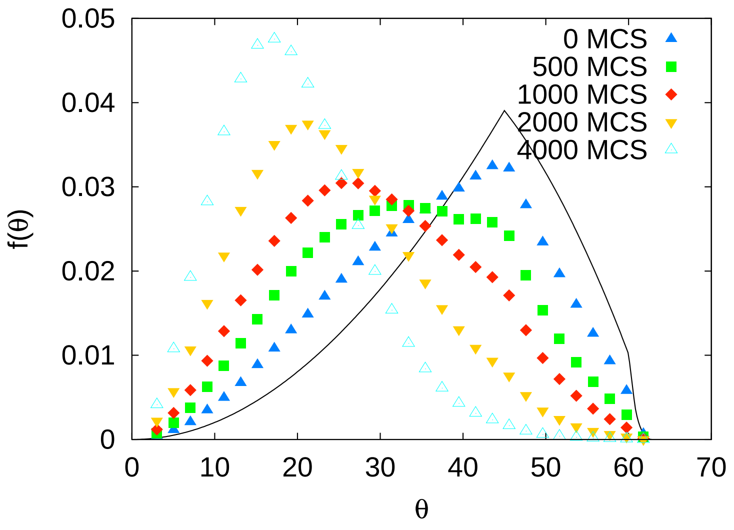
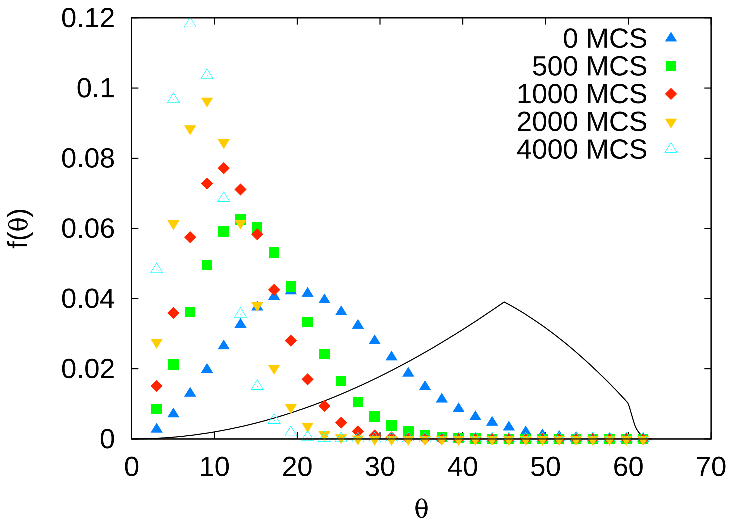


Figure 4.16: Orientation distribution function for simulation with non-random initial orientation texture with $\alpha = 4.20$ at 4000 MCS.



(a) Simulation with $\alpha = 4.20$.



(b) Simulation with $\alpha = 16.80$.

Figure 4.17: Texture weighted MDF as a function of time for simulations with non-random orientation texture.

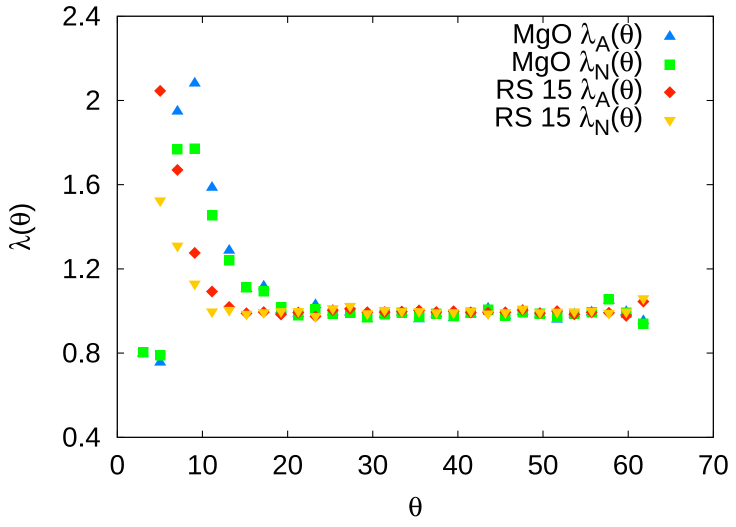
4.2.5 Simulation method validation

Here we compare our results with experimental measurements from polycrystalline magnesia samples with random orientation texture [3].

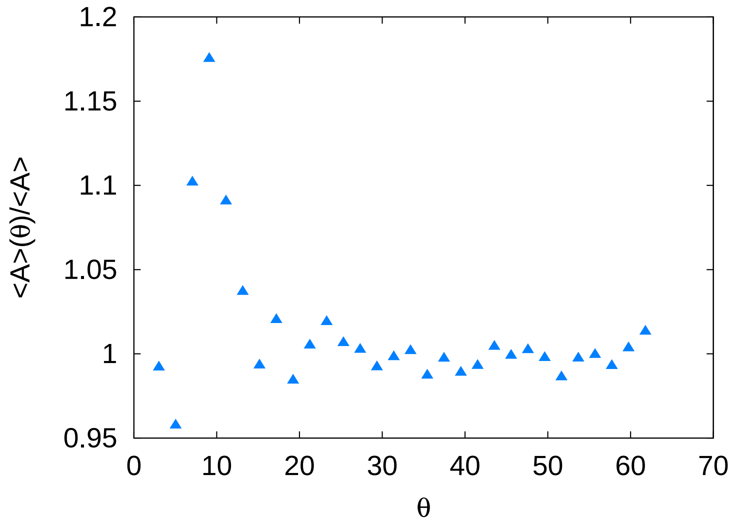
A polycrystalline, 3000ppm Ca-doped MgO sample was prepared for electron backscatter diffraction (EBSD) mapping. High purity carbonates (Alfa Aesar Puratronic $MgCO_3Mg(OH)_2$+XH_2O 99.996%, Alfa Aesar Puratronic $CaCO_3$ 99.999%) were dry-ground in an alumina mortar to promote mechanical mixing and uniform distribution of the dopant. The combined carbonates were then calcined at 1100°C for five hours in three nested MgO crucibles to avoid contamination in the furnace. The resultant powder was again dry-ground and then compacted to approximately 1000 psi in a half inch diameter cylindrical die using a Carver uniaxial press. The pellet was placed on a bed of mother powder in three nested MgO crucibles and fired using the program: 5°C/min to 900°C for 10 hours, 5°C/min to 1200°C for 7 hours, 5°C/min to 1600°C for 7 hours, 5°C/min to room temperature. Rough grinding was completed with progressively finer SiC paper, using Buehler Metadi fluid as a lubricant as water tends to degrade MgO specimens. Final polishing was accomplished using 1μm and 0.1μm diamond in oil on Buehler Mastertex cloth. The sample was then annealed at 1200°C for two hours and carbon coated (SPI-Module Carbon Coater) to eliminate charging under the electron beam.

Crystal orientation maps were obtained from a planar section using an EBSD mapping system, with 1μm spatial resolution and an average grain diameter of approximately 20μm. The dataset used in computing the MDFs included 36,223 grains and 99,420 grain boundaries.

The experimentally measured MDFs from polycrystalline MgO exhibit the same trends as our simulations, as shown in figure 4.18. Both number and area weighted MDFs are measurably non-random, with low angle boundary enhancement comparable to the simulation with $\gamma_{\mathrm{RS}}, \theta' = 15°$. Significant deviations occur in the two low angle bins, presumably due to an insufficient total number of observations (15 and 41 observations, respectively). While the absolute changes in the number and area weighted MDFs are similar, it is clear that the average area of low-angle boundaries has increased, as in figure 4.18. These results suggest that an increase in both the number and average area of grain boundaries contributes to interface texture development in real materials.



(a) Area and number weighted MDFs in multiples random distribution.



(b) Relative average grain boundary area.

Figure 4.18: Results from polycrystalline magnesia with random orientation texture (Courtesy H.M. Miller [3]).

4.3 Discussion

As previously stated, the fact that the area fraction of low energy grain boundaries increases during grain growth has been observed in a number of 2D and 3D simulations [1, 25, 27, 28, 34, 37, 120]. There are inherent difficulties in measuring the number weighted MDF for the low angle regime in microstructures with random orientation texture, as the proportion of such boundaries is only a small fraction of the total number of grain boundaries. While nearly all of the listed studies do not include measurements of the number weighted MDF, we note that Holm et al. have, and found only a "minimal increase" in the number weighted MDF during anisotropic growth [28]. Although they do not report the number of boundaries used in computing MDFs, from inspection of their microstructure at the time of computation, and assuming three boundaries per grain, a reasonable approximation is 3,750 grain boundaries. We find that the number weighted MDF appears random in measurements with fewer than 10,000 grain boundaries. Presumably, the small increase in the number weighted MDF for low energy boundaries was comparable to the noise in their measurements. Because we have found a measurable anisotropy in the number weighted MDF of a real material (figure 4.18) which is quantitatively similar to those found in our simulations, we conclude that changes in both the relative number and area of grain boundaries occur during grain growth.

Holm et al. also proposed a model for the increased length (area) of low energy boundaries driven by the requirement of interfacial equilibrium at triple junctions [28]. This model attempts to explain why low energy grain boundaries have relatively larger average areas, which the present study (figure 4.8) and several others have confirmed [1, 37, 120]. Such a model, however, does not explain why low energy boundaries also appear in greater numbers. Because the number of boundaries of any type changes only by topological events (see below) in the grain boundary network, a suitable model must depend in some way on critical event rates.

The grain boundary energy anisotropy has a much larger influence on interface texture development than mobility, which is in agreement with previous findings [31, 32, 37]. All such computations were performed on simulation domains with random orientation texture, and it is likely that given a non-random initial orientation texture, mobility anisotropy might contribute to changes in the MDF, e.g. as in abnormal growth [92]. Here we have only studied the effect of energy anisotropy on grain growth in systems with non-random orientation texture, but our results are qualitatively similar to previous work with stronger texture [28].

Finally, we note that misorientation texture development occurs without any significant effects on such microstructural features as the grain size distribution, average grain properties like the number of edges or faces, and only a minimal change in grain growth rate.

4.4 Model

It is clear from our results with non-random orientation texture that both area and number weighted MDFs evolve to approximate steady-states that are measurably different from the initial random state. In what follows we will discuss a new model for MDF evolution during grain growth. Because the number of boundaries of any type changes only by discrete topological events in the grain boundary network, our model is based on the relative rates of such critical events. We first discuss the kinds of permissible topological events that can occur during grain growth and then derive a rate equation for the change in number of grain boundaries as a function of boundary type. In this model we assume that the relative grain boundary area, which has an approximately inverse relationship to boundary energy, influences the probability of a grain boundary being eliminated by topological events. Additionally, we assume that the character of a new grain boundaries generated by critical events depends on the orientation texture of the microstructure. The combined result is a quantitative relationship between grain boundary energy, orientation texture, and the expected misorientation texture developed during grain growth. Finally, we discuss how we can use this relation to predict misorientation texture in real materials, and also solve the inverse problem, i.e. determining grain boundary energy anisotropy using misorientation texture data.

Here we use the word “area” to describe the usual measure of grain boundary size regardless of dimensionality. We will denote the area and number weighted MDFs as $f_A(\theta, t)$ and $f_N(\theta, t)$, respectively. Many other quantities, such as the area or number of boundaries, can be measured for a particular boundary type or for the entire system. To avoid introducing an excess of variables, we will always write system total functions with only a time argument. Thus the total area and number of grain boundaries in the system will be given by $A(t)$ and $N(t)$, while the total area and number of boundaries of a particular type, parameterized by θ will be $A(\theta, t)$ and $N(\theta, t)$. Likewise, the average boundary area in the system will be $\langle A\rangle(t)$, and the average area for boundary type θ

is $\langle A\rangle(\theta,t)$. Note that the following relation holds, by definition

$$\langle A\rangle(\theta,t) = \frac{A(\theta,t)}{N(\theta,t)} = \frac{f_A(\theta,t)A(t)}{f_N(\theta,t)N(t)} \tag{4.6}$$

and so the problem of predicting $f_A(\theta,t)$ and $f_N(\theta,t)$ can be reduced to a problem of predicting, for example, $\langle A\rangle(\theta,t)$ and $N(\theta,t)$. This is the approach taken here, motivated by the fact that we will later approximate $\langle A\rangle(\theta,t)$ as an explicit function of the grain boundary energy.

We focus on describing MDF development in systems with a large number of grains. By large we mean to say that statements such as "the probability of a boundary of type θ being eliminated by grain collapse" are meaningful. We will ignore the possibility of the MDF statistic ever being affected by a small number of grain boundaries in the system, or that the domain size or shape contributes in any way to the average properties of the grain boundary network. Of course the simulations presented here, as well as any experimental study, are subject to such conditions.

4.4.1 Topological events in grain growth

Any change in the number of grain boundaries in a microstructure is necessarily the result of topological changes in the grain boundary network. We will base our classification scheme of critical events on the work of Fortes and Ferro [125], who have described the possible "unit operations" that may occur during grain growth. These topological events are those that preserve the correct valencies of topological features found in a microstructure. Most importantly, any conceivable change in the topological structure of a grain boundary network can be represented as a combination of such events.

In both two and three dimensions, these events will be described as either collapse events or switching events. Collapse events are associated with the collapse of entire grains and occur when grains with three (2D) or four (3D) faces shrink to a vertex. Switching events involve grains switching topological classes (number of faces and edges) while remaining in the microstructure. These events occur by edge switching (2D), a face to edge switch (3D), or their inverses. Switching events occur in pairs and thus preserve the total number of boundaries in 2D, but occur independently in 3D. We will call the loss of a boundary by switching elimination, while the introduction of a new boundary in the microstructure will be called generation.

In what follows, the cumulative number of grain boundaries lost by grain collapse will be denoted $N_c(t)$, and the cumulative number eliminated or generated by switching events as $N_e(t)$ and $N_g(t)$, respectively. We will set each to zero at $t = 0$.

4.4.2 Critical event model

Using the above definitions, we can write an exact relation for the change in the number of boundaries of type θ during a time interval dt,

$$\begin{aligned} N(\theta, t+dt) - N(\theta, t) = p_g(\theta, t)\left[N_g(t+dt) - N_g(t)\right] \\ - p_e(\theta, t)\left[N_e(t+dt) - N_e(t)\right] - p_c(\theta, t)\left[N_c(t+dt) - N_c(t)\right]. \end{aligned} \tag{4.7}$$

Here, $p_g(\theta, t)$, $p_e(\theta, t)$, and $p_c(\theta, t)$ denote the probabilities that each possible topological event involves a boundary of type θ. The bracketed terms on the right-hand side of equation 4.7 are simply the total numbers of each critical event that occur in this time interval. Dividing by dt and taking the limit as the time interval approaches zero, we have the differential equation

$$\begin{aligned} N(t)\frac{\partial f_N(\theta, t)}{\partial t} + f_N(\theta, t)\frac{\partial N(t)}{\partial t} = p_g(\theta, t)\frac{\partial N_g(t)}{\partial t} \\ - p_e(\theta, t)\frac{\partial N_e(t)}{\partial t} - p_c(\theta, t)\frac{\partial N_c(t)}{\partial t}. \end{aligned} \tag{4.8}$$

We now define

$$\begin{aligned} \alpha_g(\theta, t) &= \frac{\partial N_g(t)}{\partial t} \\ \alpha_e(\theta, t) &= \frac{\partial N_e(t)}{\partial t} \\ \alpha_c(\theta, t) &= \frac{\partial N_c(t)}{\partial t}. \end{aligned} \tag{4.9}$$

These functions express the relative rates of each critical event type to the change in the total number of boundaries. Measuring these rates from our simulations, we find that each remains nearly constant, i.e. the rate of each type of topological event scales with the total number of grain boundaries and, similarly, the grain size. Typical values for these rates are on the order of unity. For generality, we will continue to assume each is time-dependent; we use them here merely to simplify the notation. Substituting the definitions in the previous expression into equation 4.8, we have

$$f_N(\theta,t) + N(t)\frac{\partial f_N(\theta,t)}{\partial N(t)} = \alpha_g(t)p_g(\theta,t) - \alpha_e(t)p_e(\theta,t) - \alpha_c(t)p_c(\theta,t) \tag{4.10}$$

and if the number weighted MDF reaches a steady state,

$$f_N(\theta,t) = \alpha_g(t)p_g(\theta,t) - \alpha_e(t)p_e(\theta,t) - \alpha_c(t)p_c(\theta,t) \tag{4.11}$$

from which it is clear that the steady-state number weighted MDF depends on an equilibrium of critical event rates.

Now it remains to determine an adequate form for each probability function. Our first assumption, which is satisfied quite generally by our simulations, is simply

$$p_g(\theta,t) = f_0(\theta,t) \tag{4.12}$$

where $f_0(\theta,t)$ is the texture weighted misorientation distribution function (TMDF), as defined in a previous section. This assumption implies that when new boundaries are generated, they appear between grains that have a relationship given by the texture weighted MDF but have no other correlation.

Next we assume a functional form for the elimination and collapse events. Both types of events can only eliminate a boundary that already exists in the system. We therefore expect that when the number of a particular boundary type θ is N and the probability of elimination is P, if the number is increased to $2N$ then its probability of elimination is $2P$, i.e. $p_e(\theta,t)$ and $p_c(\theta,t)$ should both be first order homogenous in $f_N(\theta,t)$. We further assume that these probabilities depend on the average areas $\langle A\rangle(\theta,t)$ of boundary types. That the smallest boundaries on a grain are the first to be eliminated by topological switching was postulated decades ago by C.S. Smith [126]. The probability of elimination therefore should decrease with increasing average area. Likewise, if boundaries with relatively large areas persist on a grain as it loses faces, they should be the most likely to be eliminated by the grain collapse events. The probability of their elimination by collapse should then increase with increasing average area.

Because the observed values of $\langle A\rangle(\theta,t)$ occur only within a small range of $\langle A\rangle(t)$, we approximate each boundary elimination probability by a series expansion in $\langle A\rangle(\theta,t)$ about $\langle A\rangle(t)$. The probabilities must be invariant with uniform scaling of the domain size, therefore each term in the expansion must be divided by an appropriate scale factor such as $\langle A\rangle(t)$. For $p_e(\theta,t)$ we take

$$p_e(\theta,t) = \frac{f_N(\theta,t)}{\langle A\rangle(t)} \sum_{i=0}^{\infty} a_{e,i} \left[\langle A\rangle(\theta,t) - \langle A\rangle(t)\right]^i . \tag{4.13}$$

Note that, in isotropic grain growth and more generally, when $\langle A\rangle(\theta,t) = \langle A\rangle(t)$, the above should reduce to $f_N(\theta,t)$, so that

$$a_{e,0} = \langle A\rangle(t). \tag{4.14}$$

Similarly, for $p_c(\theta,t)$ we take

$$p_c(\theta,t) = \frac{f_N(\theta,t)}{\langle A\rangle(t)} \sum_{i=0}^{\infty} a_{c,i} \left[\langle A\rangle(\theta,t) - \langle A\rangle(t)\right]^i \tag{4.15}$$

and have that

$$a_{c,0} = \langle A\rangle(t). \tag{4.16}$$

4.4.3 Comparison with simulations

Now we apply our model to the special case of determining the steady-state number and area fraction weighted MDFs. After some algebraic manipulations, we have the first-order approximation

$$f_N(\theta,t) = f_0(\theta,t) \left[u(t) + v(t)\frac{\langle A\rangle(\theta,t)}{\langle A\rangle(t)}\right]^{-1} \tag{4.17}$$

where we have set

$$u(t) = \frac{\alpha_e(t)(1-a_{e,1}) + \alpha_c(t)(1-a_{c,1})}{\alpha_g(t)} \tag{4.18}$$

and

$$v(t) = \frac{\alpha_e(t)a_{e,1} + \alpha_c(t)a_{c,1}}{\alpha_g(t)}. \tag{4.19}$$

In our simulations we find that $u(t)$ takes values around 5/3, and $v(t)$ of about $-2/3$, so that in much of the applicable range of $\langle A\rangle(\theta, t)$, the bracket on the right hand side of equation 4.17 is linear in $\langle A\rangle(\theta, t)/\langle A\rangle(t)$. Thus, to first order,

$$f_N(\theta, t) \approx f_0(\theta, t)\frac{\langle A\rangle(\theta, t)}{\langle A\rangle(t)} \tag{4.20}$$

or, in multiples of a random distribution (MRD),

$$\lambda_N(\theta, t) \approx \frac{\langle A\rangle(\theta, t)}{\langle A\rangle(t)} \tag{4.21}$$

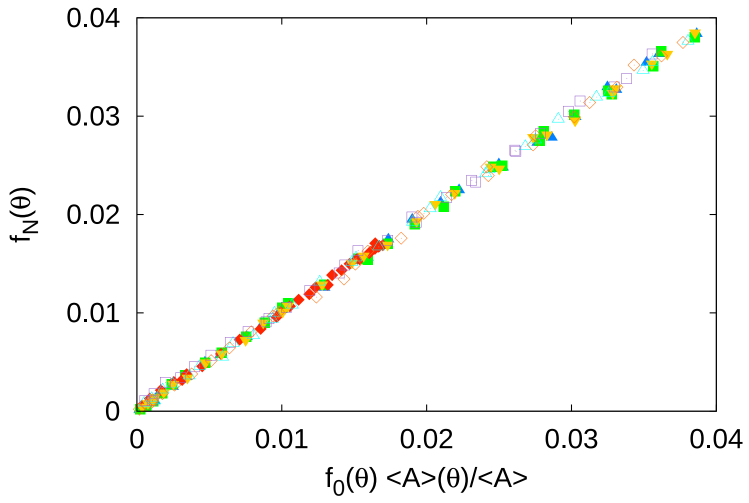
where the appropriate proportionality constant for both equations can be determined by normalizing equation 4.20.

The local value of the number weighted MDF is plotted against the right hand side of equation 4.20 in figure 4.19 for each simulation at 2000 MCS. For both 3D and 2D simulations, the result is a scatter of points lying close to the line $f(x) = x$. This result confirms the quite general applicability of equation 4.20 in predicting the steady-state number weighted MDF. We note that the initial microstructure in all simulations satisfies equation 4.20 trivially, since we begin with an isotropically grown polycrystal and randomly assigned orientations.

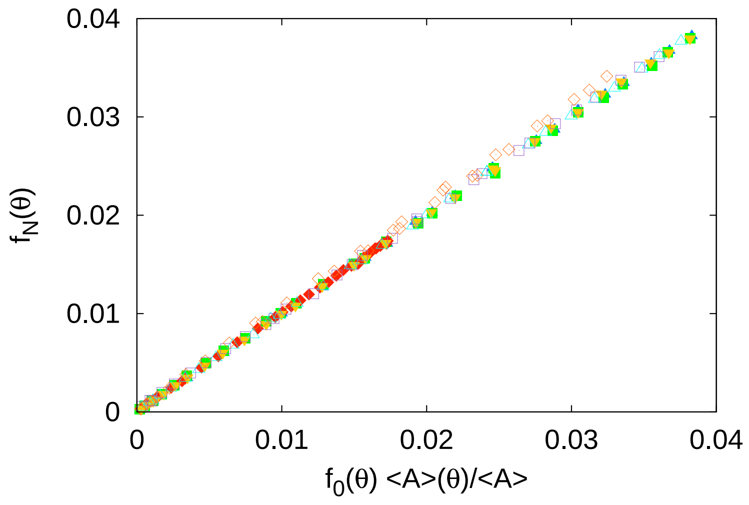
It is certainly possible to generate a microstructure where the average boundary area and the grain boundary number fraction are not related by equation 4.20. We then expect that in such cases the number weighted MDF transitions to the steady state according to equation 4.10. However, in our simulations the critical event mechanism appears to operate quickly enough to compensate for changes in the average areas of grain boundaries, leading to the steady-state equation being satisfied at all times. Thus none of the Monte Carlo simulations performed here can be used to test the time-dependent part of the critical event model.

Using equations 4.6 and 4.20, we can also compute the steady state area weighted MDF as

$$f_A(\theta, t) \approx f_0(\theta, t)\left[\frac{\langle A\rangle(\theta, t)}{\langle A\rangle(t)}\right]^2 \tag{4.22}$$



(a) 2D Monte Carlo simulations.



(b) 3D Monte Carlo simulations.

Figure 4.19: Scatter plot of left- and right-hand sides of equation 4.20 for all Monte Carlo simulations with random orientation texture, 2000 MCS.

or, in multiples random

$$\lambda_A(\theta, t) \approx \left[\frac{\langle A\rangle(\theta, t)}{\langle A\rangle(t)}\right]^2 \tag{4.23}$$

Again, a suitable proportionality constant can be computed by normalization of equation 4.22. We plot the area weighted MDF measured in each simulation against the right hand side of equation 4.22 in figure 4.20. Similar to the result above, we have points lying about a straight line. While the area weighted MDF now depends on the square of $\langle A\rangle(\theta, t)$, deviations from this line are not significantly larger than those in figure 4.19. Again we find that equation 4.22 is satisfied for all simulation times, regardless of whether the area weighted MDF has reached a steady state.

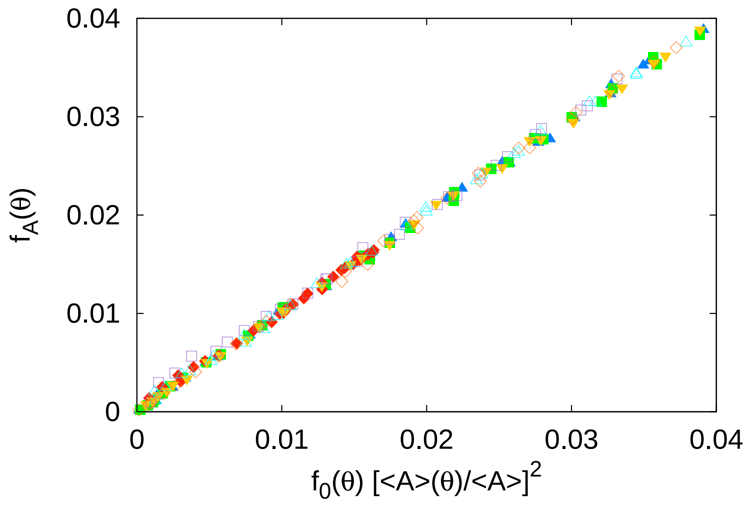
We note that the above relations explain why the area and number weighted MDFs in figure 4.7 had slopes related by a factor of 2. In fact, the steady state equations derived from the critical event model imply that $\ln \lambda_A \approx 2 \ln \lambda_N$, which is plotted in figure 4.21.

Although the number and area weighted MDFs progress through a transient state before reaching their steady state distributions, equations 4.20 and 4.22 appear to be satisfied at all times. This is demonstrated in figure 4.22, where we take the particular case of the 3D Monte Carlo simulation with Read-Shockley energy, $\theta' = 45°$, and cubic crystal symmetry. It then also is possible that the steady-state equation is satisfied, at least approximately, in simulations with non-random orientation texture. In fact, this appears to happen quite generally, as shown in figure 4.23. Clearly, the mechanism of misorientation texture development by critical events occurs on a smaller time scale than the mechanisms controlled by boundary lengthening and orientation texture development.

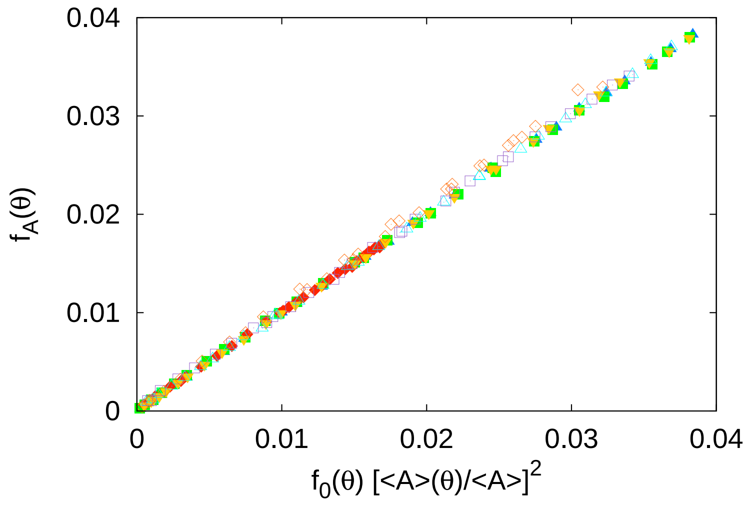
We do not know whether the magnesia sample has reached a steady state MDF or ODF. However, the results above suggest that equation 4.20 might also apply in this case. In fact we find that the relation is satisfied quite well, as shown in figure 4.24.

4.4.4 Boundary lengthening model

Aside from the TMDF, the essential input to the critical event model presented above is the average area for each boundary type. A quantitative model for average boundary area has been presented by Holm et al. [28]. They consider a two-dimensional geometry where the ends of a grain boundary with energy $\gamma(\theta)$ meet in triple lines with two other boundaries having energy $\gamma_{\max}$, the maximum grain boundary energy in the system, as shown in figure 4.25. To begin, the boundaries are in



(a) 2D Monte Carlo simulations.



(b) 3D Monte Carlo simulations.

Figure 4.20: Scatter plot of left- and right-hand sides of equation 4.22 for all Monte Carlo simulations, 2000 MCS.

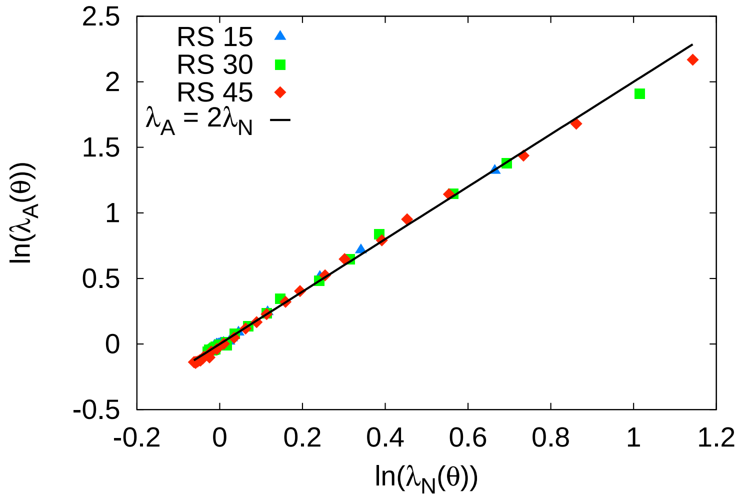


Figure 4.21: Relation between area and number weighted MDFs for several 3D simulations, 500 MCS.

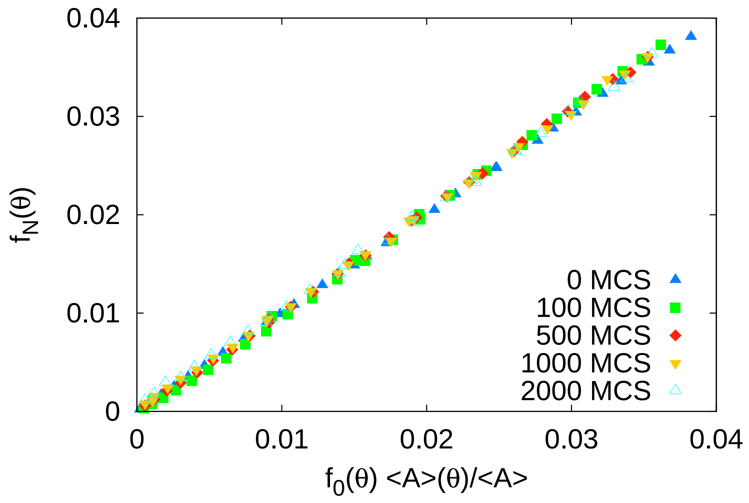


Figure 4.22: Scatter plot of left- and right-hand sides of equation 4.20. 3D Monte Carlo simulations with $\gamma_{\mathrm{RS}}(\theta)$, $\theta' = 45°$ and cubic crystal symmetry.

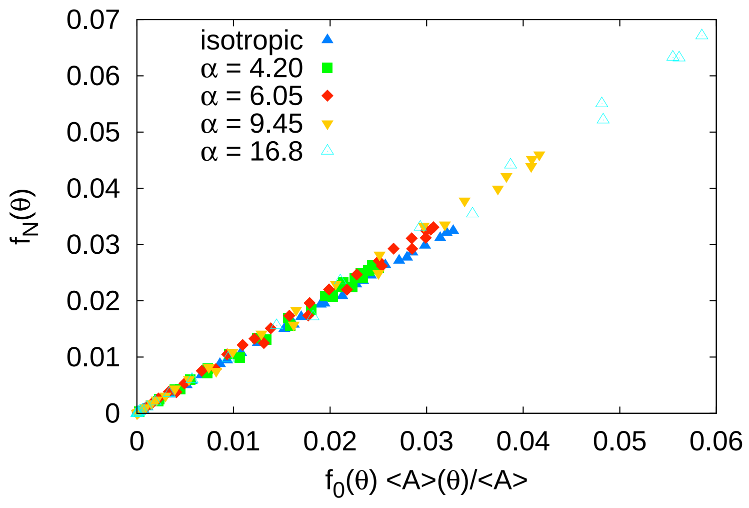


Figure 4.23: Scatter plot of left- and right-hand sides of equation 4.20. 3D Monte Carlo simulations with non-random orientation texture.

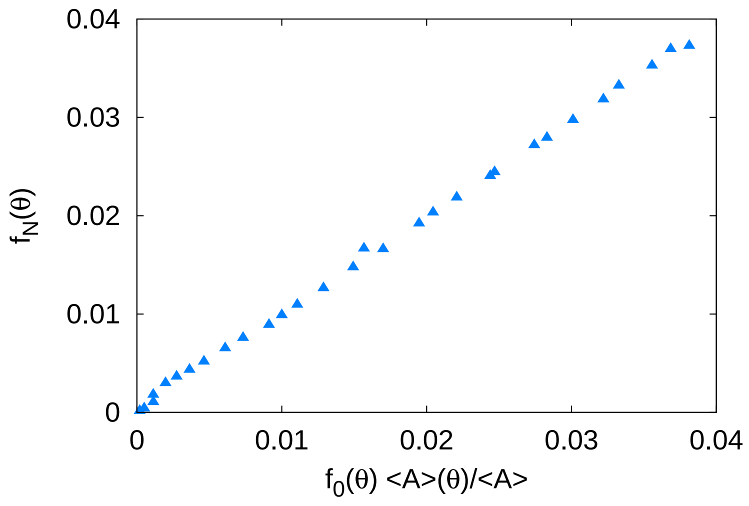


Figure 4.24: Scatter plot of left- and right-hand sides of equation 4.20 for polycrystalline magnesia.

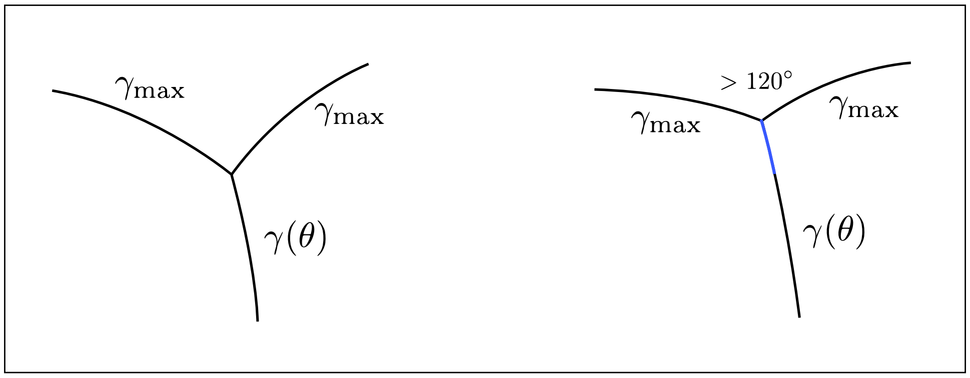


Figure 4.25: Schematic of grain boundary lengthening process. A boundary with energy $\gamma(\theta)$ intersects two boundaries with the maximum grain boundary energy in an isotropic configuration. Boundary lengthening occurs as the boundaries adjust to satisfy mechanical equilibrium at the triple junction.

an isotropic configuration, i.e. all boundaries have the same length L and every dihedral angle is 120°. Now we let the boundaries relax until the equilibrium triple line geometry is obtained. The boundary with energy $\gamma(\theta)$ now has length $L + \Delta L(\theta)$. Holm et al. used a linear approximation for the boundary lengthening, i.e.

$$\frac{\Delta L}{L} = 1 + \alpha(1 - \gamma(\theta)) \tag{4.24}$$

where α is a constant fitting parameter. Note that equation 4.24 was derived under the condition that $\gamma_{\max} = 1$. However, we could easily compute the exact lengthening in such a case as

$$\frac{\Delta L}{L} = \frac{1}{2}\left[1 - \frac{\sqrt{3}}{\tan\cos^{-1}\frac{\gamma(\theta)}{2\gamma_{\max}}}\right] \tag{4.25}$$

This approach does not require a fitting parameter. In both models, the boundary lengthening that occurs for this particular geometry is assumed to be representative of the average boundary lengthening throughout the system. For two dimensions, the increase in boundary area should be proportional to the boundary lengthening given by either equation 4.24 or 4.25. Three dimensional area changes should follow the square of this relation.

Each boundary lengthening model predicts the general trends of average boundary area with energy as measured from our simulations. Figure 4.26 shows that the average boundary area is a nonlinear function of energy in both two and three dimensions, especially in the low energy regime.

Equation 4.24 does not accurately predict the average boundary areas in regions where the data curve upward, e.g. for the lowest energy grain boundaries, regardless of the chosen fitting parameter. We find that the “exact” lengthening model in equation 4.25 is even less accurate, and in fact exhibits curvature in the opposite direction.

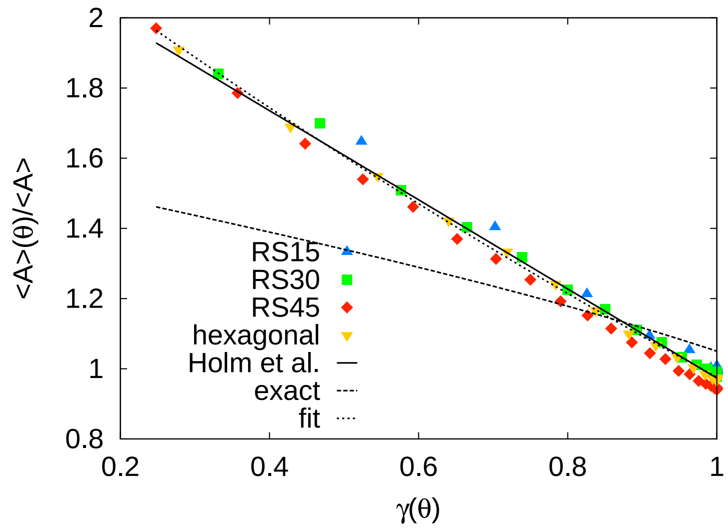
Several approximations used in the boundary lengthening model may lead to its inaccuracy. First, the topological neighborhood of any grain boundary is certainly more complex than that used, i.e. with every boundary under consideration intersecting only boundaries of the system average energy. A more accurate form of the model might consider the boundary lengthening processof a given boundary type in a number of neighborhoods with boundaries of various energies. Additionally, the three boundaries meeting at a triple junction must satisfy the geometric constraint that $\Delta g_{\mathrm{AB}}\Delta g_{\mathrm{BC}} = \Delta g_{\mathrm{CA}}$ (or any permutation of A, B, and C). For disorientation angles, this implies that $\theta_{AB} \leq \theta_{BC} + \theta_{CA}$ (for all permutations). In particular, two low angle grain boundaries meeting at a triple junction necessarily meet with a third low angle boundary, while triple junctions with high angle grain boundaries are not as constrained. This condition implies that, in general, low angle grain boundaries should not lengthen as much as the model boundary lengthening model predicts. This is in fact true of our simulations, see e.g. the average area data from the simulation with a step function energy (figure 4.8).

Although the boundary lengthening process is simple, it appears to be quite difficult to derive a simple quantitative model to predict its resulting effect on a microstructure. However, the relation between grain boundary energy and average area is approximately one to one, and so we can at least fit an empirical formula to it. With a second order polynomial fit

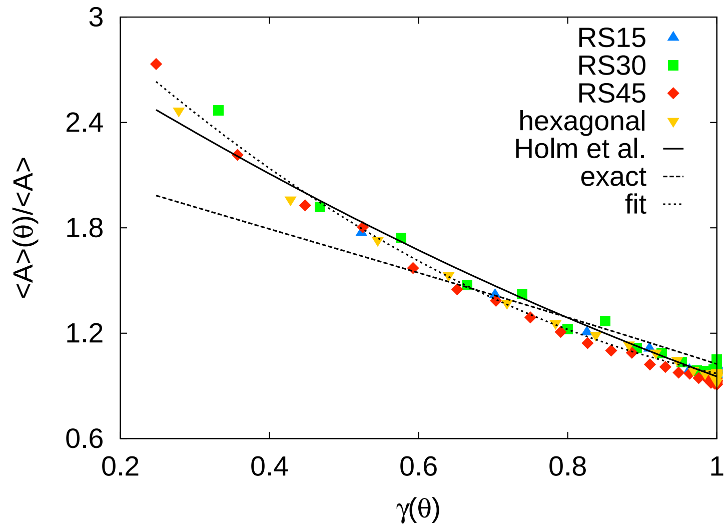
$$\frac{\langle A\rangle(\theta)}{\langle A\rangle} = a + b\frac{\gamma}{\gamma_{\mathrm{max}}} + c\left(\frac{\gamma}{\gamma_{\mathrm{max}}}\right)^2 \tag{4.26}$$

we find for 2D $a = 2.345$, $b = -1.592$, and $c = 0.2231$ with $\chi^2 = 0.0005238$, while for 3D $a = 3.610$, $b = -4.369$, and $c = 1.729$ with $\chi^2 = 0.001475$. In particular, we can easily use equation 4.26 to approximate any of the steady state MDFs from simulations with random orientation texture, as shown in figure 4.27. The accuracy of the result is very good, especially for low angle grain boundaries.

We note that, because each of these functions are invertible within the domain of possible bound-



(a) 2D Monte Carlo simulations, 1000 MCS.



(b) 3D Monte Carlo simulations, 500 MCS.

Figure 4.26: Scatter plot of grain boundary energy and average boundary area for Monte Carlo simulations. Lines indicate boundary lengthening model predictions and polynomial fit.

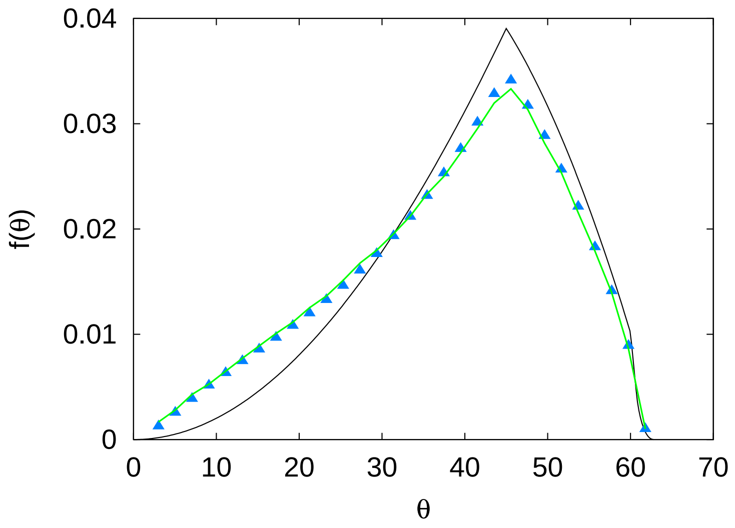


Figure 4.27: Steady state area weighted MDF computed using the second order polynomial fit 4.26, $\gamma_{\mathrm{RS}}, \theta' = 45°$. Solid line is the corresponding area weighted MDF from simulation, 500 MCS.

ary energies $[0, \gamma_{\max}]$, there is a possibility to use them in the context of deriving energy functions from measured misorientation texture data. For example, we consider the area and number weighted MDFs from the polycrystalline magnesia sample mentioned above. Figure 4.28 shows the results of a fit to $\gamma(\theta)$ using equations 4.26 with the 3D fitting parameters. We see that the majority of the points follow a Read-Shockley type function with $\theta' = 15°$, as expected.

4.5 Summary

We have performed simulations of 2D and 3D grain growth with anisotropic energy and mobility, and both random and non-random orientation texture. Grain boundary energy anisotropy was found to produce a measurable effect on both number and area weighted MDFs in all cases. The average area and the number of relatively low energy grain boundaries increases relative to higher energy grain boundaries. The effect of mobility anisotropy on interface texture appears to be insignificant, in agreement with previous results. Both area-weighted and number-weighted MDFs develop into steady-state distributions after some initial transient. Similar results are obtained in all simulations regardless of dimensionality, crystal symmetry, or the form of the energy function. Measured number and area weighted MDFs in polycrystalline MgO are qualitatively similar to those produced by

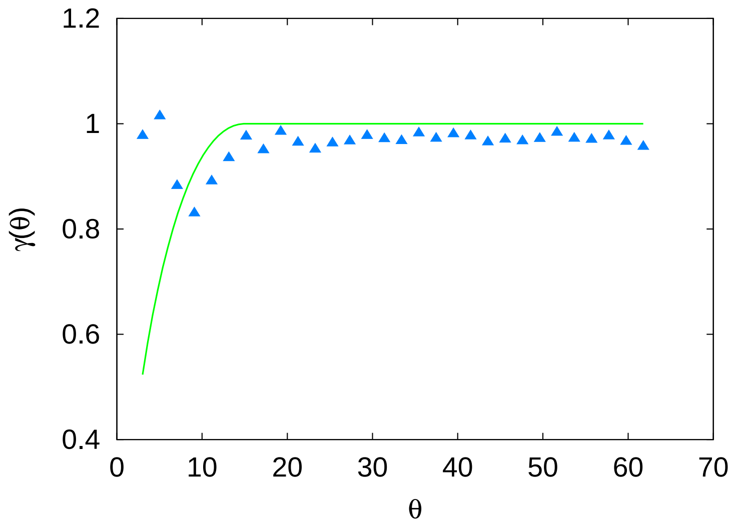


Figure 4.28: Energy function derived from MDF measurements of polycrystalline magnesia. Solid line is $\gamma_{\mathrm{RS}}, \theta' = 15°$.

simulations with anisotropic energy. When the initial orientation texture of the microstructure is non-random and boundary properties are anisotropic, we find that the texture weighted MDF is not constant, and becomes an additional mechanism for interface texture development.

To describe our results, we have proposed a critical event model for the evolution of number and area weighted misorientation distribution functions during grain growth. This model demonstrates the explicit dependence of misorientation texture on grain boundary energy anisotropy and orientation texture through the texture weighted misorientation distribution function. Predictions from the model are compared to area and number weighted MDFs measured in Monte Carlo simulations with random orientation texture and anisotropic interfacial properties. The steady state equation of our model appears to be a good fit to all data, indicating that the critical event mechanism works on a finer time scale than boundary lengthening or orientation texture development.

Chapter 5

Inclination dependent anisotropy

In the last chapter, we found that misorientation dependent energy anisotropy influences the average area of grain boundaries, which in turn leads to nonrandom number and area weighted MDFs. Because boundary properties were a function of misorientation only, any single grain boundary had a unique, constant energy and mobility, and moreover, the boundaries of a given type could be counted. When the boundary properties are inclination (interface normal) dependent, this is no longer the case, as the normal vector changes continuously over curved interfaces. The generation or elimination of a particular boundary type may occur as a result of boundary motion, even without topological events. Clearly, predicting the GBCD that results from grain growth with inclination dependent anisotropy is a much different problem than modelling the effects of misorientation dependent anisotropy.

In this chapter we will demonstrate that energy and mobility anisotropy have similar effects on the resulting interface texture, in that the texture depends most strongly on the energy function. Likewise, many of the other qualitative aspects of misorientation texture development apply here as well.

Energy α	Energy $\mathbf{n}'$	$\gamma_{\max}$ / $\gamma_{\min}$	Mobility α	Mobility $\mathbf{n}'$	$M_{\max}$ / $M_{\min}$
0	$\langle 1,0,0\rangle$	1	0	$\langle 1,0,0\rangle$	1
0.08281	$\langle 1,0,0\rangle$	1.07	0	$\langle 1,0,0\rangle$	1
0.05915	$\langle 1,1,1\rangle$	1.05	0	$\langle 1,0,0\rangle$	1
0.1774	$\langle 1,1,1\rangle$	1.15	0	$\langle 1,0,0\rangle$	1
0.2957	$\langle 1,1,1\rangle$	1.25	0	$\langle 1,0,0\rangle$	1
0.8872	$\langle 1,1,1\rangle$	1.75	0	$\langle 1,0,0\rangle$	1
0	$\langle 1,0,0\rangle$	1	13.60	$\langle 1,1,1\rangle$	12.5
0.2957	$\langle 1,1,1\rangle$	1.25	13.60	$\langle 1,1,1\rangle$	12.5

Table 5.1: Summary of simulation parameters for grain growth with inclination dependent anisotropy.

5.1 Simulations

Grain boundary properties are assigned on the basis of macroscopic grain boundary geometry. The interfacial energy $\gamma(\Delta\mathbf{g}, \mathbf{n})$ and $M(\Delta\mathbf{g}, \mathbf{n})$ functions are defined by an interface plane scheme described earlier. We write the interface properties as a function of the interface normal vectors with respect to both grains, i.e.

$$\gamma(\Delta\mathbf{g}, \mathbf{n}_A) = \gamma_s(\mathbf{n}_A) + \gamma_s(\Delta\mathbf{g}\,\mathbf{n}_A) \tag{5.1}$$

and

$$M(\Delta\mathbf{g}, \mathbf{n}_A) = M_s(\mathbf{n}_A) + M_s(\Delta\mathbf{g}\,\mathbf{n}_A) \tag{5.2}$$

where $\gamma_s(\mathbf{n})$ and $M_s(\mathbf{n})$ are invariant under the relevant crystal symmetry operators. These functions will take the form

$$\gamma_s(\mathbf{n}) = 1 + \alpha \min_{\mathbf{O}_i \in \mathcal{O}_h} \left\| \frac{\mathbf{O}_i\mathbf{n}}{\|\mathbf{O}_i\mathbf{n}\|} - \frac{\mathbf{n}'}{\|\mathbf{n}'\|} \right\|_2 \tag{5.3}$$

with $M_s(\mathbf{n})$ similarly defined. Here, α is a positive constant and $\mathbf{n}'$ is a fixed crystal direction, usually taken to be either $\langle 1,0,0\rangle$ or $\langle 1,1,1\rangle$. $\mathbf{O}_i$ represents an operator from the point group $\mathcal{O}_h$, as we consider only cubic crystal symmetry. Note that minima for either function occur for normal vectors near $\mathbf{n}'$ and symmetrically equivalent directions. Also, the value of α can be used to control the magnitude of the energy and mobility anisotropy. The functions are isotropic when $\alpha = 0$.

We perform a number of simulations with various choices for α and $\mathbf{n}'$. Table 5.1 contains a summary of our choices for each.

Additionally, we performed one simulation with the following function for γ_s.

$$\gamma_s(\mathbf{n}) = 1 + \frac{1}{8}\left[1 + \tanh\left(12 \min_{\mathbf{O}_i \in \mathcal{O}_h} \left\| \frac{\mathbf{O}_i\mathbf{n}}{\|\mathbf{O}_i\mathbf{n}\|} - \frac{\mathbf{n}'}{\|\mathbf{n}'\|}\right\|_2 - 5\right)\right] \tag{5.4}$$

where $\mathbf{n}' = \langle 1, 1, 1\rangle$. This function is similar in magnitude and form to that described above with $\gamma_{\max}/\gamma_{\min} = 1.25$, but has broader, "flattened" minima and maxima due to the hyperbolic tangent function.

5.2 Results

5.2.1 General observations

As in simulations with misorientation dependent anisotropy, grain growth kinetics with inclination dependent mobility anisotropy are nearly identical to the isotropic case, figure 4.1. The kinetic exponent n for isotropic growth is found to be $n = 0.99$. Again, the absolute rate of grain growth with mobility anisotropy is different because the average mobility of boundaries in such cases is greater than one.

While the simulations with anisotropic mobility showed an increased rate of grain growth, we find that those with anisotropic energy actually had slower growth and a grain growth exponent of only about $n = 0.94$. The energy functions used here did not have strong anisotropy, and so the average boundary energy in such cases was nearly one. To explain the decrease in growth rate, we might consider that the inclination dependent anisotropy leads to changes in the local shape of boundaries. Boundaries may become more planar in the vicinity of low energy inclinations in an attempt to reduce total energy. These boundaries would then have less curvature and thus a smaller average driving force for grain boundary motion. This seems reasonable, as the growth rate diminishes with greater anisotropy.

While the typical shapes of grain boundaries appear to change with inclination dependent energy anisotropy, grain shapes still appear to be mostly equiaxed, figure 4.3.

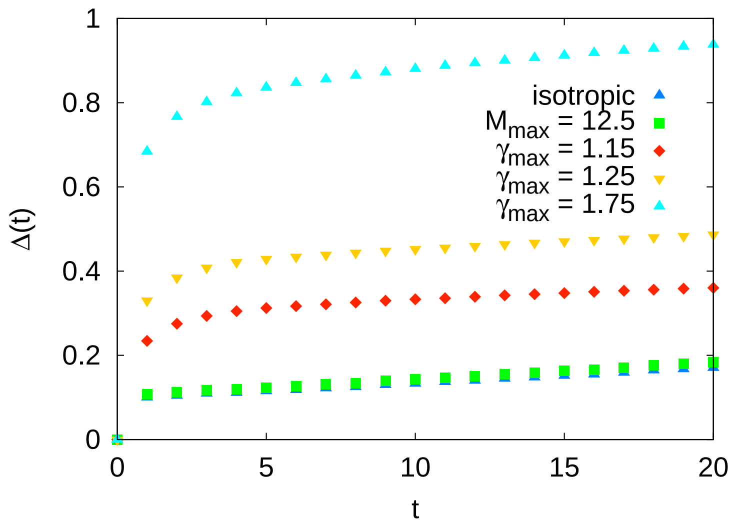


Figure 5.1: The function $\Delta(t)$ for select simulations. Data from isotropic simulation are obscured by that from simulation with anisotropic mobility.

5.2.2 Time dependence

We found that misorientation dependent anisotropy led to steady state MDFs, and the same is true here. We redefine $\Delta(t)$ as a function of the GBCD, using the invariant measures on the sphere and in $SO(3)$,

$$\Delta(t) = \int \|f(\varphi_1, \Phi, \varphi_2, \theta, \phi, t) - f(\varphi_1, \Phi, \varphi_2, \theta, \phi, 0)\| \, dV \tag{5.5}$$

where the volume element dV is given by

$$dV = d\varphi_1 d\sin(\Phi) d\varphi_2 d\sin(\theta) d\phi. \tag{5.6}$$

This relation is plotted as a function of time for several simulations in figure 5.1. In each case, $\Delta(t)$ initially increases quickly and later slows. This typically occurs after the number of grains in the sample has decreased by more than about 30%, or equivalently, after the average grain volume has increased by a factor of nearly 1.5. At later times, the rate of change of Δ for all simulations is of the same magnitude as for the isotropic case. This suggests that $\Delta(t)$ is changing only because of increasing sampling noise.

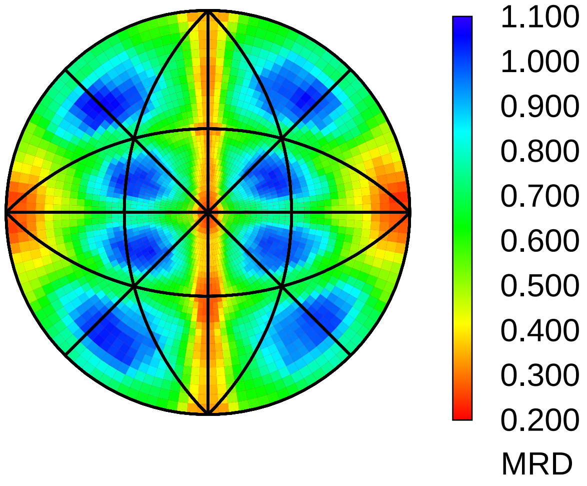
5.2.3 Property dependence

In figures 5.2 and 5.3 we have plotted the energy and grain boundary population for several fixed misorientations for simulations with anisotropic interfacial energy. There are clearly maxima (minima) in each energy plot corresponding to minima (maxima) in the related grain boundary distribution. When the grain boundary energy is anisotropic, low energy boundaries have relatively high populations and high energy boundaries occur less frequently. This trend is consistent across all simulations, and shows that texture development with inclination dependent energy follows the same qualitative behavior as with misorientation dependent anisotropy. Figure 5.4 shows the average population of grain boundaries as a function of energy for various simulations with anisotropic energy. There is a clear trend for grain boundaries with low energy to have relatively larger populations that those with higher energy.

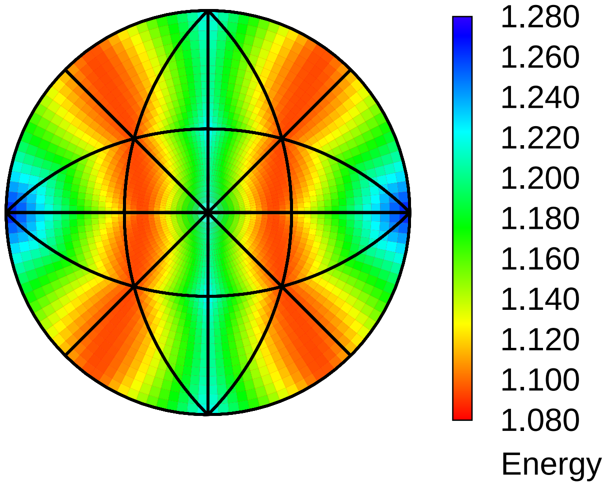
The increase in low energy boundary planes in a microstructure agrees qualitatively with the solution of a shrinking embedded grain. However, we observe that the steady-state GBCDs measured here are nonzero everywhere, i.e. there exist grain boundaries of all types in each data set. This is in constrast to the embedded grain, which typically has missing inclinations. Figure 5.5 shows the steady-state population of interface normals of an initally spherical shrinking grain, which can be directly compared to the result in figure 5.2. This result suggests that the grain boundary network connectivity contributes in some way to the grain boundaries being unable to attain their minimum energy configuration.

Figures 5.6 and 5.7 show the GBCD at fixed misorientation with anisotropic mobility and isotropic energy. When the energy is isotropic, the distribution of grain boundary planes is random, regardless of the mobility. Again, this result in qualitatively the same as that found with misorientation dependent properties. Note that while there is some deviation from the exact random distribution with anisotropic mobility, this deviation is no larger than that measured in isotropic growth.

We find that the energy to area mapping is nonunique, as shown in figure 5.8. Here we have plotted a two dimensional histogram of boundaries categorized by their energy and measured population in the simulation with $\gamma_{\min}/\gamma_{\max} = 1.25$, $\mathbf{n'} = \langle 1, 1, 1 \rangle$. While the relationship between energy and population in simulations with misorientation dependent anisotropy was only approximately one-to-

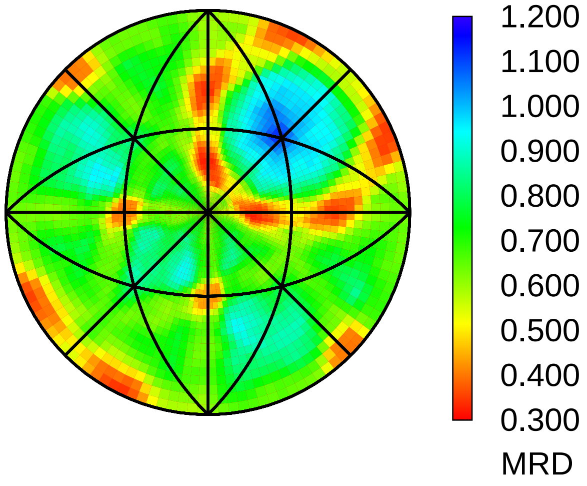


(a) Grain boundary population in multiples random.

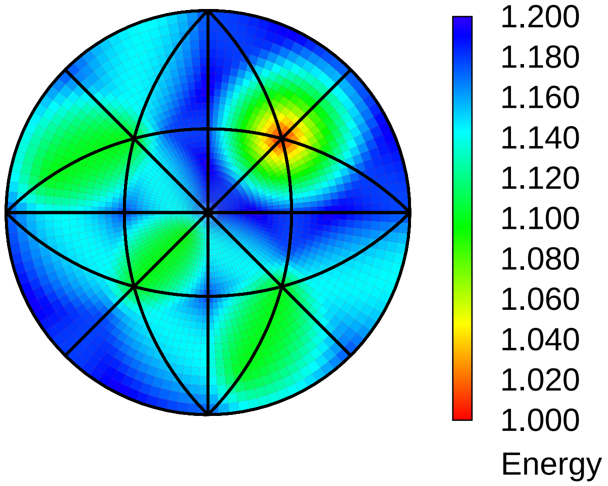


(b) Grain boundary energy (arbitrary units).

Figure 5.2: Grain boundary population and energy for a fixed misorientation of 45° about $\langle 1, 0, 0\rangle$, where $\gamma_{\text{min}}/\gamma_{\text{max}} = 1.25$, $\mathbf{n}' = \langle 1, 1, 1\rangle$.



(a) Grain boundary population in multiples random.



(b) Grain boundary energy (arbitrary units).

Figure 5.3: Grain boundary population and energy for a fixed misorientation of 60° about $\langle 1,1,1\rangle$, where $\gamma_{\min}/\gamma_{\max} = 1.25$, $\mathbf{n}' = \langle 1,1,1\rangle$.

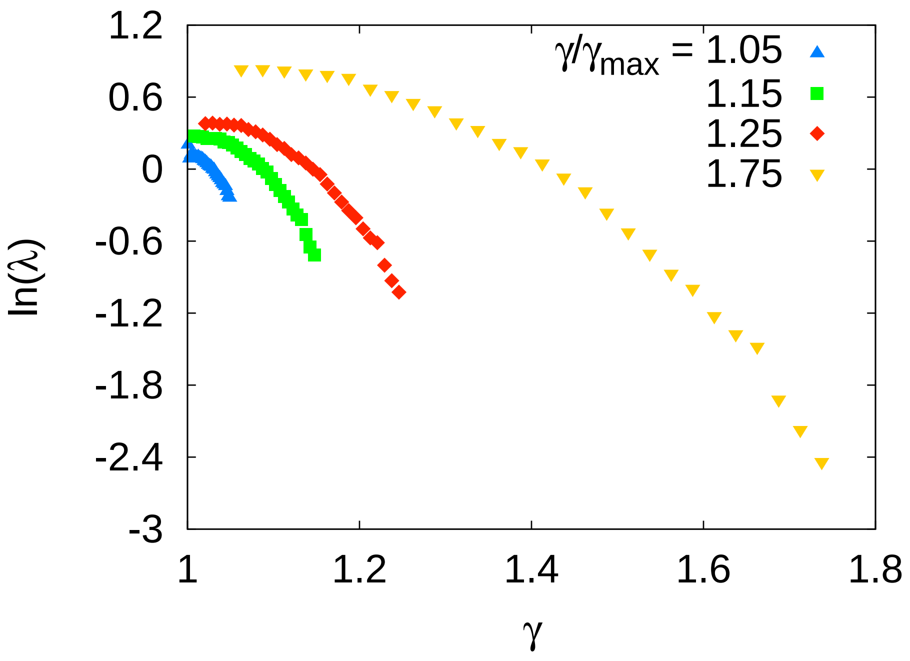


Figure 5.4: Average population of grain boundaries as a function of grain boundary energy γ for various simulations.

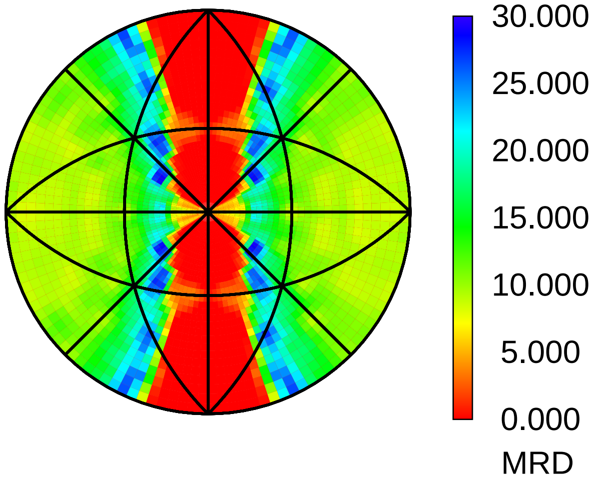
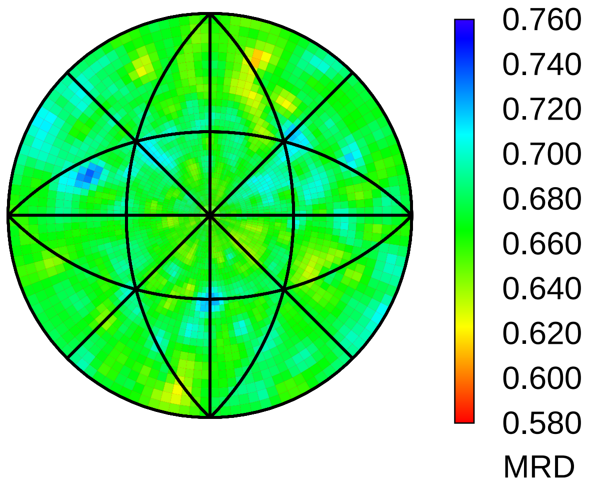
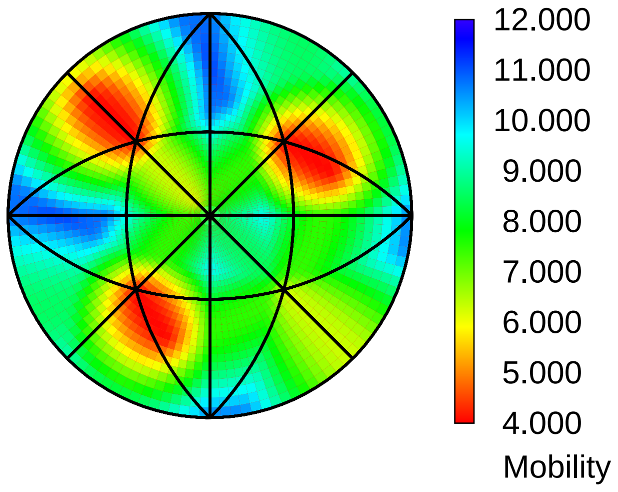


Figure 5.5: Population of interface normals on an embedded shrinking grain, with misorientation 45° about $\langle 1, 0, 0 \rangle$. Compare with the result of figure 5.2.

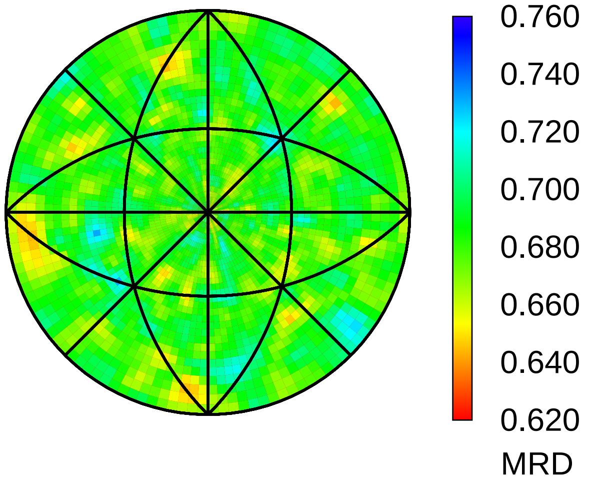


(a) Grain boundary population in multiples random.

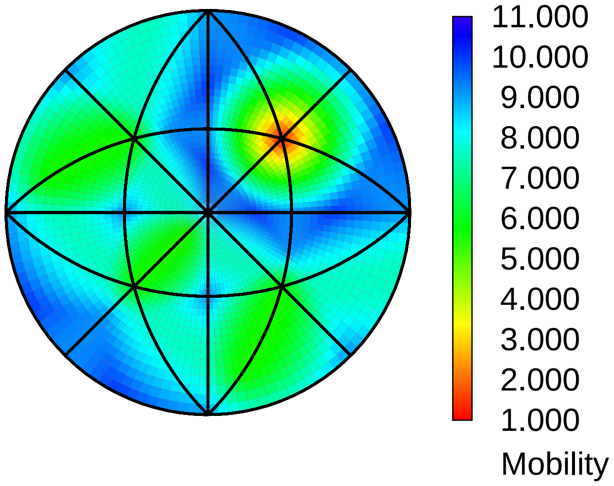


(b) Grain boundary mobility (arbitrary units).

Figure 5.6: Grain boundary population and mobility for a fixed misorientation of 45° about $\langle 1, 1, 0 \rangle$, where $M_{\min}/M_{\max} = 12.5$, $\mathbf{n}' = \langle 1, 1, 1 \rangle$.



(a) Grain boundary population in multiples random.



(b) Grain boundary mobility (arbitrary units).

Figure 5.7: Grain boundary population and energy for a fixed misorientation of 60° about $\langle 1,1,1 \rangle$, where $M_{\min}/M_{\max} = 12.5$, $\mathbf{n'} = \langle 1,1,1 \rangle$.

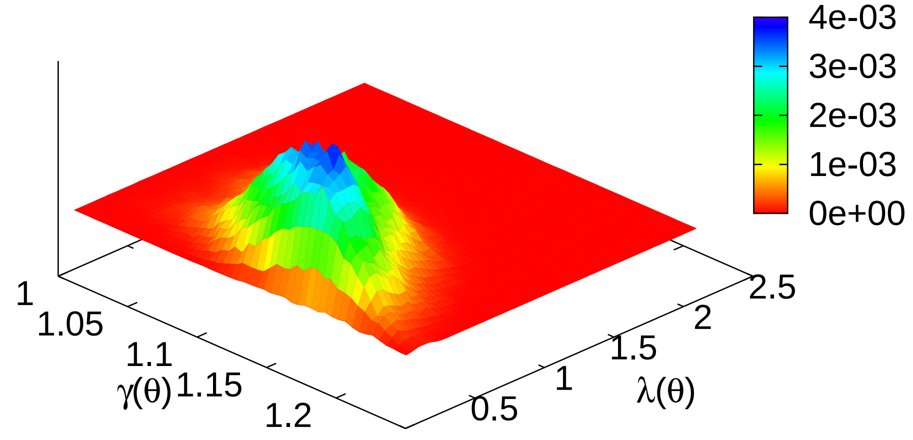


Figure 5.8: Frequency of individual boundary types with energy γ and population λ (multiples random).

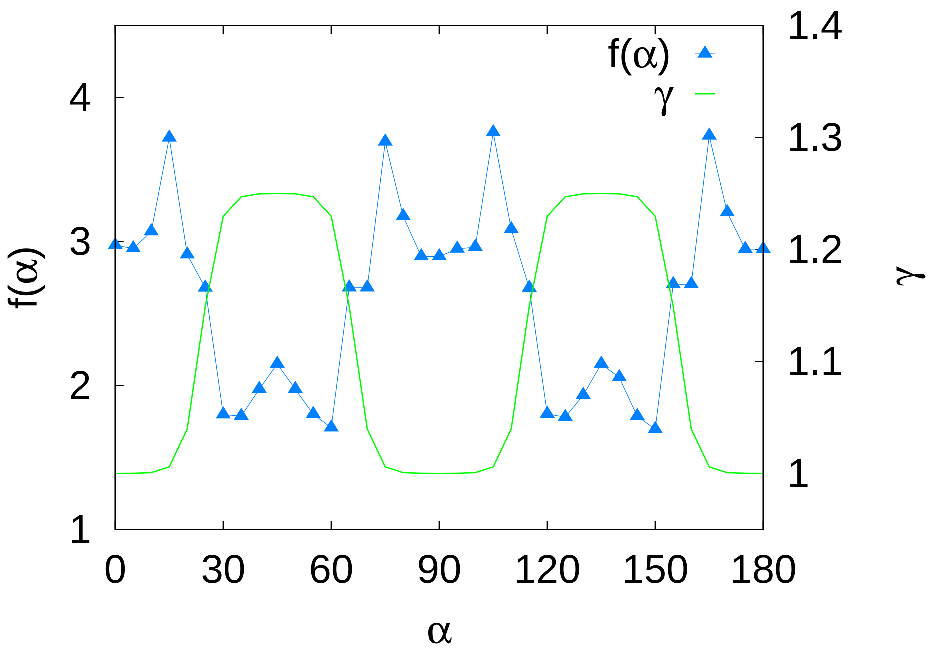


Figure 5.9: Population and energy of grain boundaries with $< 10°$ misorientation, as a function of angle α about the zone [001]. Simulated data from simulation with "flat" energy wells.

one, here the non-uniqueness is much more pronounced. In particular, grain boundaries with the same misorientation and identical energy values might have significantly different populations, as shown in figure 5.9. Here the only significant difference between boundaries with the same energy is the local energy gradient. Note that this effect occurs for both high and low energy boundaries. This effect has also been observed experimentally by Saylor et al. [4], as shown in figure 5.10.

5.2.4 Simulation method validation

The grain boundary distributions from simulations are very similar to those measured in previous studies of magnesia. Figure 5.11 compares plots of boundary plane distributions for various fixed misorientations obtained with the GRAIN3D simulations with equivalent plots of measured distributions. The success of the simulations in this respect is probably a result of the fact that energy anisotropy alone accounts for the development of anisotropic grain boundary plane distributions.

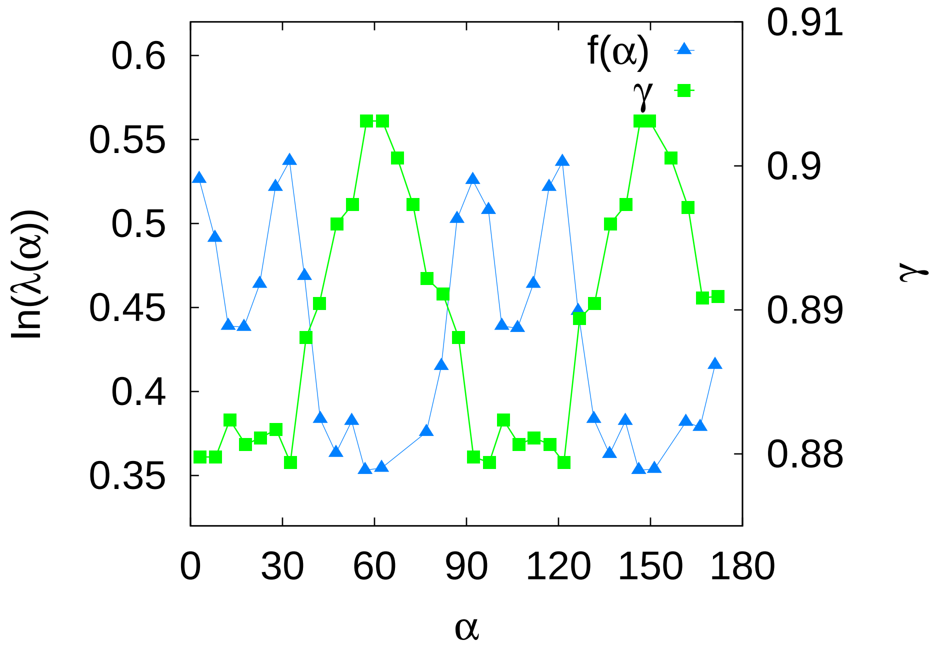
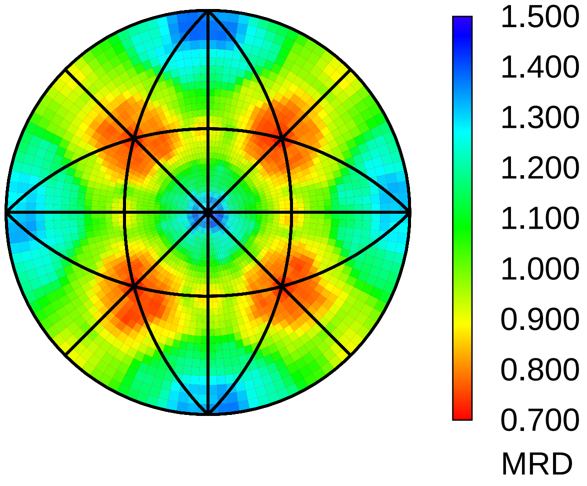
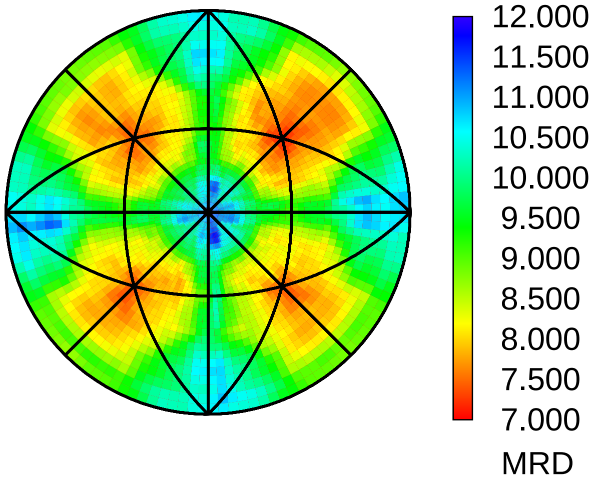


Figure 5.10: Experimentally measured population and energy of grain boundaries in polycrystalline magnesia, as a function of angle α about the zone [010], from Saylor et al. [4].



(a) Simulation at $t = 10$.



(b) Experimental data [6].

Figure 5.11: Comparison of low angle grain boundary distributions for simulation with [100] type energy minima and experimental data from polycrystalline magnesia.

5.3 Discussion

We hypothesized an inverse relation between grain boundary energy and grain boundary distributions. In figures 5.2 and 5.3 we have plotted the energy and simulated grain boundary population for several fixed misorientations for simulations with anisotropic interfacial energy.

There are clearly maxima (minima) in each energy plot corresponding to minima (maxima) in the related grain boundary distribution. When the grain boundary energy is anisotropic, low energy boundaries have relatively high populations and high energy boundaries occur less frequently. This trend is consistent across all simulations, and shows that texture development with inclination dependent energy follows the same qualitative behavior as with misorientation dependent anisotropy.

It is reasonable now to investigate the relationship between grain boundary energy and average boundary area, and compare this to the relation found in the last chapter. First, we find that the energy to area mapping is nonunique, as shown in figures 5.8 and 5.9. While this was the case with misorientation dependent anisotropy, here it is much more pronounced. The only significant difference between these simulations is the overall magnitude of the energy anisotropy.

However, this implies that the energy gradients are also different. With inclination dependent energy anisotropy, the triple junction equilibrium depends on a balance of both the magnitude and the gradient of the energy of each boundary. We might then expect that for such a mapping to be even approximately one-to-one, it should at least consider both the energy and the local energy neighborhood of grain boundary planes.

For the systems examined here, the independence of the grain boundary character distribution from the mobility is noteworthy. It should be emphasized that even though the mobility anisotropy used was ten times larger than the energy anisotropy, it had a negligible effect on the grain boundary character distribution. In the absence of a strong orientation texture, it is possible that the highest mobility boundaries move through grains quickly and are then replaced with randomly generated boundary types, reducing the population of high mobility boundaries. However, the results presented here demonstrate that this in not the case. It is then likely that in this context it is important to recognize that when the mobility is anisotropic and the energy is isotropic, then the condition for equilibrium at the triple junctions requires that grain boundaries adopt orientations such that the grain boundary dihedral angles are all equal. Therefore, as long as the orientations of the triple lines are randomly distributed, the grain boundary plane orientations will also be randomly distributed. On the other hand, when the boundary energy is anisotropic, the boundaries planes at triple junctions adjust to low energy orientations that also satisfy the interfacial equilibrium constraint and this produces a relatively higher population of low energy boundaries. Finally, it should be noted that we do not necessarily expect the grain boundary character distribution to be independent of mobility when significant orientation texture is present.

5.4 Summary

We have performed simulations of grain growth with inclination dependent energy and mobility anisotropy and random orientation texture. Grain boundary distributions under these conditions appear to reach steady states after some initial period. The distribution of grain boundary planes for a fixed misorientation is dependent on the energy function for that misorientation, and specifically, an inverse relation exists for local extrema. Boundary plane distributions exhibit relative minima (maxima) for planes at energy maxima (minima), which is consistent with experimental observations and other simulated results. The assumed grain boundary mobility anisotropy is shown to have no measurable effect on grain boundary plane distributions under the simulation conditions used.

Chapter 6

Conclusions and Future Work

In this work, we have studied grain growth in two and three dimensions using a variety of computational methods. Our primary objective has been to model the quantitative relationships between the anisotropy of interfacial properties and the resulting interface texture. These models, along with their assumptions, accuracy, and limitations, have been discussed in the previous two chapters.

Regardless of the details of individual simulations, several distinct trends emerge. The introduction of grain boundary energy anisotropy leads to interface texture, while mobility anisotropy has a much weaker effect. For grain growth with misorientation dependent energy anisotropy and random orientation texture, the resulting interface texture can be approximated reasonably well by a function of grain boundary energy alone. Non-random initial orientation textures and anisotropic energy allow for the possibility of orientation texture development during grain growth, which greatly influences the resulting misorientation distribution. Inclination dependent anisotropy increases the complexity of both the energy functions and resulting grain boundary texture. However, the overall trend towards increasing the ratio of low energy to high energy boundaries remains true. In the case of misorientation texture dependent anisotropy, the mechanism of interface texture development appears to be due to a combination of boundary lengthening and biased critical events. The average length or area of boundaries becomes non-uniform as a result of triple junction equilibration, while these average areas dictate the rate at which different types of boundaries are eliminated. With inclination dependent energy anisotropy, the triple junction equilibrium condition depends on local energy gradients in addition to the magnitude of the energy. This leads to a more complex model for average boundary area.

As for future work, several problems must be addressed before the models presented here can be used to accurately predict grain boundary texture development or to infer grain boundary energy anisotropy from experiment. One problem is to develop a better model for the relationship between grain boundary type and average area, as well as to extend the current model to the more general five parameter case. This appears to be the most critical limiting factor in the present model, as the proposed relation between average area and number or area weighted MDFs is highly accurate. Another significant task is to confirm that the results and models presented here agree with experiment. Along with a better quantitative relationship, this will require a complete measure of grain boundary energy anisotropy over all macroscopic parameters, as well as an accurate measurements of the GBCD and the ODF.

Many other issues remain. In particular, this work has presented results from only a limited subset of the possible choices for grain boundary energy and mobility anisotropy, as well as orientation texture. There is reason to believe that extreme anisotropy might lead to effects that cannot be explained by our model, e.g. grain boundary faceting. Whether our models are applicable to a large or small subset of engineering materials can only be answered through further simulation and experiment.

Bibliography

[1] J Gruber, DC George, AP Kuprat, GS Rohrer, and AD Rollett. Effect of anisotropic grain boundary energy on grain boundary distributions during grain growth. *Mat. Sci. Forum*, 467-470:733–738, 2004.

[2] G Hasson, JY Boos, I Herbeuval, M Biscondi, and C Goux. Theoretical and experimental determination of grain boundary structures and energies – correlation with various experimental results. *Surface Sci.*, 31:115–137, 1972.

[3] HM Miller. unpublished work. 2007.

[4] DM Saylor, A Morawiec, and GS Rohrer. The relative free energies of grain boundaries in magnesia as a function of five macroscopic parameters. *Acta Mater.*, 51(13):3675–3686, 2003.

[5] DM Saylor, A Morawiec, and GS Rohrer. Distribution and energies of grain boundaries in magnesia as a function of five degrees of freedom. *J. Am. Ceram. Soc.*, 85(12):3081–3083, 2002.

[6] DM Saylor, A Morawiec, and GS Rohrer. Distribution of grain boundaries in magnesia as a function of five macroscopic parameters. *Acta Mater.*, 51(13):3663–3674, 2003.

[7] GS Rohrer, DM Saylor, B El Dasher, BL Adams, AD Rollett, and P Wynblatt. The distribution of internal interfaces in polycrystals. *Z. Metallk.*, 95(4):197–214, 2004.

[8] DM Saylor, BS El Dasher, AD Rollett, and GS Rohrer. Distribution of grain boundaries in aluminum as a function of five macroscopic parameters. *Acta Mater.*, 52(12):3649–3655, 2004.

[9] DM Saylor, B El Dasher, T Sano, and GS Rohrer. Distribution of grain boundaries in srtio3 as a function of five macroscopic parameters. *J. Am. Ceram. Soc.*, 87(4):670–676, 2004.

[10] DM Saylor, B El Dasher, Y Pang, HM Miller, P Wynblatt, AD Rollett, and GS Rohrer. Habits of grains in dense polycrystalline solids. *J. Am. Ceram. Soc.*, 87(4):724–726, 2004.

[11] CS Kim, Y Hu, GS Rohrer, and V Randle. Five parameter grain boundary distribution in grain boundary engineered brass. *Scr. Mater.*, 52(7):633–637, 2005.

[12] V Randle, Y Hu, GS Rohrer, and CS Kim. Distribution of misorientations and grain boundary planes in grain boundary engineered brass. *Mat. Sci. Tech.*, 21(11):1287–1292, 2005.

[13] GS Rohrer, V Randle, CS Kim, and Y Hu. Changes in the five parameter grain boundary character distribution in alpha-brass brought about by iterative thermomechanical processing. *Acta Mater.*, 54(17):4489–4502, 2006.

[14] B Alexandreanu, B Capell, and GS Was. Combined effect of special grain boundaries and grain boundary carbides on IGSCC of Ni-16Cr-9Fe-XC alloys. *Mat. Sci. Engr. A*, 300(1-2):94–104, 2001.

[15] B Alexandreanu and GS Was. Grain boundary deformation-induced intergranular stress corrosion cracking of Ni-16Cr-9Fe in 360 degrees c water. *Corrosion*, 59(8):705–720, 2003.

[16] GO Williams, V Randle, JR Cowan, and P Spellward. The role of misorientation and phosphorus content on grain growth and intergranular fracture in iron-carbon-phosphorus alloys. *J. Micro.*, 213:321–327, 2004.

[17] U Krupp, PEG Wagenhuber, WM Kane, and CJ McMahon. Improving resistance to dynamic embrittlement and intergranular oxidation of nickel based superalloys by grain boundary engineering type processing. *Mat. Sci. Tech.*, 21(11):1247–1254, 2005.

[18] B Alexandreanu and GS Was. The role of stress in the efficacy of coincident site lattice boundaries in improving creep and stress corrosion cracking. *Scr. Mater.*, 54(6):1047–1052, 2006.

[19] G Owen and V Randle. On the role of iterative processing in grain boundary engineering. *Scr. Mater.*, 55(10):959–962, 2006.

[20] CS Chung, JK Kim, HK Kim, and WJ Kim. Improvement of high-cycle fatigue life in a 6061 Al alloy produced by equal channel angular pressing. *Mat. Sci. Engr. A*, 337(1-2):39–44, 2002.

[21] Y Gao, M Kumar, RK Nalla, and RO Ritchie. High cycle fatigue of nickel based superalloy ME3 at ambient and elevated temperatures: Role of grain boundary engineering. *Metal. Mat. Trans. A*, 36A(12):3325–3333, 2005.

[22] V Randle and H Davies. Evolution of microstructure and properties in alpha-brass after iterative processing. *Metal. Mat. Trans. A*, 33(6):1853–1857, 2002.

[23] B Alexandreanu, BH Sencer, V Thaveeprungsriporn, and GS Was. The effect of grain boundary character distribution on the high temperature deformation behavior of Ni-16Cr-9Fe alloys. *Acta Mater.*, 51(13):3831–3848, 2003.

[24] T Furuhara and T Maki. Grain boundary engineering for superplasticity in steels. *J. Mat. Sci.*, 40(4):919–926, 2005.

[25] DC Hinz and JA Szpunar. Modeling the effect of coincidence site lattice boundaries on grain growth textures. *Phys. Rev. B*, 52(14):9900–9909, 1995.

[26] K Mehnert and P Klimanek. Monte carlo simulation of grain growth in textured metals using anisotropic grain boundary mobilities. *Comp. Mat. Sci.*, 7(1-2):103–108, 1996.

[27] N Ono, K Kimura, and T Watanabe. Monte carlo simulation of grain growth with the full spectra of grain orientation and grain boundary energy. *Acta Mater.*, 47(3):1007–1017, 1999.

[28] EA Holm, GN Hassold, and MA Miodownik. On misorientation distribution evolution during anisotropic grain growth. *Acta Mater.*, 49(15):2981–2991, 2001.

[29] MC Demirel, AP Kuprat, DC George, GK Straub, and AD Rollett. Linking experimental characterization and computational modeling of grain growth in Al-Foil. *Interface Sci.*, 10(2-3):137–141, 2002.

[30] A Kazaryan, Y Wang, SA Dregia, and BR Patton. Grain growth in systems with anisotropic boundary mobility: Analytical model and computer simulation. *Phys. Rev. B*, 6318(18), 2001.

[31] A Kazaryan, Y Wang, SA Dregia, and BR Patton. Grain growth in anisotropic systems: Comparison of effects of energy and mobility. *Acta Mater.*, 50(10):2491–2502, 2002.

[32] A Kazaryan, BR Patton, SA Dregia, and Y Wang. On the theory of grain growth in systems with anisotropic boundary mobility. *Acta Mater.*, 50(3):499–510, 2002.

[33] MC Demirel, AP Kuprat, DC George, and AD Rollett. Bridging simulations and experiments in microstructure evolution. *Phys. Rev. Lett.*, 90(1), 2003.

[34] GN Hassold, EA Holm, and MA Miodownik. Accumulation of coincidence site lattice boundaries during grain growth. *Mat. Sci. Tech.*, 19(6):683–687, 2003.

[35] D Kinderlehrer, I Livshits, GS Rohrer, S Ta'asan, and P Yu. Mesoscale simulation of the evolution of the grain boundary character distribution. In *Recrystallization and Grain Growth, Parts 1 and 2*, volume 467-470, pages 1063–1068, 2004.

[36] K Barmak, WE Archibald, J Kim, CS Kim, AD Rollett, GS Rohrer, S Ta'asan, and D Kinderlehrer. Grain boundary energy and grain growth in highly-textured al films and foils: Experiment and simulation. In *Icotom 14: Textures of Materials, Parts 1 and 2*, volume 495-497, pages 1255–1260, 2005.

[37] J Gruber, DC George, AP Kuprat, GS Rohrer, and AD Rollett. Effect of anisotropic grain boundary properties on grain boundary plane distributions during grain growth. *Scr. Mater.*, 53(3):351–355, 2005.

[38] K Barmak, J Kim, CS Kim, WE Archibald, GS Rohrer, AD Rollett, D Kinderlehrer, S Ta'asan, H Zhang, and DJ Srolovitz. Grain boundary energy and grain growth in al films: Comparison of experiments and simulations. *Scr. Mater.*, 54(6):1059–1063, 2006.

[39] C Hammond. *The Basics of Crystallography and Diffraction.* International Union of Crystallography: Oxford University Press, Oxford, 1997.

[40] JF Nye. *Physical Properties of Crystals: Their Representation by Tensors and Matrices.* Clarendon, Oxford, 1957.

[41] A Morawiec. *Orientations and Rotations: Computations in Crystallographic Textures.* Springer, New York, 2004.

[42] HJ Bunge. *Texture Analysis in Materials Science.* Butterworths, Boston, 1982.

[43] JB Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality.* Princeton University Press, Princeton, 1999.

[44] V Randle. *The Role of the Coincidence Site Lattice in Grain Boundary Engineering.* Institute of Materials, London, 1996.

[45] AP Sutton and Balluffi RW. *Interfaces in Crystalline Materials.* Clarendon, Oxford, 1995.

[46] WT Read and W Shockley. Dislocation models of crystal grain boundaries. *Phys. Rev.*, 78(3):275–289, 1950.

[47] D Wolf. Structure-energy correlation for grain boundaries in fcc metals 2. boundaries on the (110) and (113) planes. *Acta Metall.*, 37(10):2823–2833, 1989.

[48] D Wolf. Structure-energy correlation for grain boundaries in fcc metals 1. boundaries on the (111) and (100) planes. *Acta Metall.*, 37(7):1983–1993, 1989.

[49] D Wolf. Structure and energy of general grain boundaries in bcc metals. *J. Appl. Phys.*, 69(1):185–196, 1991.

[50] D Wolf. Effect of interatomic potential on the calculated energy and structure of high-angle coincident site grain boundaries 2. (100) twist boundaries in cu, ag and au. *Acta Metall.*, 32(5):735–748, 1984.

[51] D Wolf. Effect of interatomic potential on the calculated energy and structure of high-angle coincident site grain boundaries 1. (100) twist boundaries in aluminum. *Acta Metall.*, 32(2):245–258, 1984.

[52] Y Huang and FJ Humphreys. Subgrain growth and low angle boundary mobility in aluminium crystals of orientation 110001. *Phys. Rev. B*, 74(11), 2006.

[53] G Gottstein and Shvindlerman LS. *Grain Boundary Migration in Metals: Thermodynamics, Kinetics, Applications*. CRC Press, Boca Raton, 1999.

[54] D Turnbull. Theory of grain boundary migration rates. *Trans. Am. Inst. Min. Metal. Engr.*, 191(8):661–665, 1951.

[55] H Zhang, DJ Srolovitz, JF Douglas, and JA Warren. Characterization of atomic motion governing grain boundary migration. *Phys. Rev. B*, 74(11), 2006.

[56] K Kawasaki, T Nagai, and K Nakashima. Vertex models for two dimensional grain growth. *Phil. Mag. B*, 60(3):399–421, 1989.

[57] T Nagai, K Fuchizaki, and K Kawasaki. Orientation effect on grain growth. *Phys. A*, 204(1-4):450–463, 1994.

[58] M Enomoto, M Kamiya, and T Nagai. Computer simulation of two-dimensional grain growth with anisotropic grain boundary energy and mobility by vertex model. In *Grain Growth In Polycrystalline Materials II, Pts 1 and 2*, pages 71–82, 1996.

[59] C Maurice and J Humphreys. 2- and 3-d curvature driven vertex simulations of grain growth. In *ICGG-3: Third International Conference on Grain Growth*, pages 81–90, 1998.

[60] D Weygand, Y Brechet, and J Lepinoux. Influence of a reduced mobility of triple points on grain growth in two dimensions. *Acta Mater.*, 46(18):6559–6564, 1998.

[61] D Weygand, Y Brechet, and J Lepinoux. A vertex dynamics simulation of grain growth in two dimensions. *Phil. Mag. B*, 78(4):329–352, 1998.

[62] D Weygand, Y Brechet, J Lepinoux, and W Gust. Three dimensional grain growth: a vertex dynamics simulation. *Phil. Mag. B*, 79(5):703–716, 1999.

[63] D Weygand, Y Brechet, and J Lepinoux. Zener pinning and grain growth: a two dimensional vertex computer simulation. *Acta Mater.*, 47(3):961–970, 1999.

[64] D Weygand, Y Brechet, and J Lepinoux. On the nucleation of recrystallization by a bulging mechanism: a two dimensional vertex simulation. *Phil. Mag. B*, 80(11):1987–1996, 2000.

[65] D Weygand, Y Brechet, and J Lepinoux. Mechanisms and kinetics of recrystallisation: a two dimensional vertex dynamics simulation. *Interface Sci.*, 9(3-4):311–317, 2001.

[66] D Weygand, Y Brechet, and J Lepinoux. A vertex simulation of grain growth in 2d and 3d. *Adv. Engr. Mat.*, 3(1-2):67–71, 2001.

[67] C Herring. Surface tension as a motivation for sintering. In WE Kingston, editor, *The Physics of Powder Metallurgy*, pages 143–179. McGraw-Hill, New York, 1951.

[68] HJ Frost, CV Thompson, and DT Walton. Simulation of thin film grain structures 1. grain growth stagnation. *Acta Metall. Mat.*, 38(8):1455–1462, 1990.

[69] HJ Frost, CV Thompson, and DT Walton. Simulation of thin film grain structures 2. abnormal grain growth. *Acta Metall. Mat.*, 40(4):779–793, 1992.

[70] HJ Frost. Microstructural evolution in thin films. *Mat. Char.*, 32(4):257–273, 1994.

[71] HJ Frost and CV Thompson. Computer simulation of grain growth. *Curr. Op. Sol. St. Mat. Sci.*, 1(3):361–368, 1996.

[72] SP Riege, CV Thompson, and HJ Frost. Simulation of the influence of particles on grain structure evolution in two dimensional systems and thin films. *Acta Mater.*, 47(6):1879–1887, 1999.

[73] SPA Gill and ACF Cocks. A variational approach to two dimensional grain growth 2. numerical results. *Acta Mater.*, 44(12):4777–4789, 1996.

[74] SPA Gill and ACF Cocks. A short note on a variational approach to normal grain growth. *Scr. Mater.*, 35(1):9–12, 1996.

[75] MP Anderson, DJ Srolovitz, GS Grest, and PS Sahni. Computer simulation of grain growth 1. kinetics. *Acta Metall.*, 32(5):783–791, 1984.

[76] DJ Srolovitz, MP Anderson, GS Grest, and PS Sahni. Computer simulation of grain growth 3. influence of a particle dispersion. *Acta Metall.*, 32(9):1429–1438, 1984.

[77] MP Anderson, GS Grest, and DJ Srolovitz. Grain growth in 3 dimensions - a lattice model. *Scr. Metal.*, 19(2):225–230, 1985.

[78] DJ Srolovitz, GS Grest, and MP Anderson. Computer simulation of grain growth 5. abnormal grain growth. *Acta Metall.*, 33(12):2233–2247, 1985.

[79] MP Anderson, GS Grest, RD Doherty, K Li, and DJ Srolovitz. Inhibition of grain growth by 2nd phase particles - 3 dimensional monte carlo computer simulations. *Scr. Metal.*, 23(5):753–758, 1989.

[80] MP Anderson, GS Grest, and DJ Srolovitz. Computer simulation of normal grain growth in 3 dimensions. *Phil. Mag. B*, 59(3):293–329, 1989.

[81] AD Rollett, DJ Srolovitz, and MP Anderson. Simulation and theory of abnormal grain growth anisotropic grain boundary energies and mobilities. *Acta Metall.*, 37(4):1227–1240, 1989.

[82] GS Grest, MP Anderson, DJ Srolovitz, and AD Rollett. Abnormal grain growth in 3-dimensions. *Scr. Metal. Mat.*, 24(4):661–665, 1990.

[83] GN Hassold, EA Holm, and DJ Srolovitz. Effects of particle size on inhibited grain growth. *Scr. Metal. Mat.*, 24(1):101–106, 1990.

[84] AD Rollett, DJ Srolovitz, MP Anderson, and RD Doherty. Computer simulation of recrystallization 3. influence of a dispersion of fine particles. *Acta Metall. Mat.*, 40(12):3475–3495, 1992.

[85] EA Holm, DJ Srolovitz, and JW Cahn. Microstructural evolution in 2-dimensional 2-phase polycrystals. *Acta Metall. Mat.*, 41(4):1119–1136, 1993.

[86] GN Hassold and DJ Srolovitz. Computer simulation of grain growth with mobile particles. *Scr. Metal. Mat.*, 32(10):1541–1547, 1995.

[87] T Baudin, P Paillard, and R Penelle. Grain growth simulation starting from experimental data. *Scr. Mater.*, 36(7):789–794, 1997.

[88] EA Holm, N Zacharopoulos, and DJ Srolovitz. Nonuniform and directional grain growth caused by grain boundary mobility variations. *Acta Mater.*, 46(3):953–964, 1998.

[89] NM Hwang. Simulation of the effect of anisotropic grain boundary mobility and energy on abnormal grain growth. *J. Mat. Sci.*, 33(23):5625–5629, 1998.

[90] T Baudin, P Paillard, and R Penelle. Simulation of the anisotropic growth of goss grains in Fe_3%Si sheets. *Scr. Mater.*, 40(10):1111–1116, 1999.

[91] AD Rollett. Texture development dependence on grain boundary properties. In *Textures Of Materials, Pts 1 and 2*, pages 251–256, 2002.

[92] EA Holm, MA Miodownik, and AD Rollett. On abnormal subgrain growth and the origin of recrystallization nuclei. *Acta Mater.*, 51(9):2701–2716, 2003.

[93] AD Rollett. Crystallographic texture change during grain growth. *JOM*, 56(4):63–68, 2004.

[94] ADF Rollett. Crystallographic texture change during grain growth. *Jom*, 56(4):63–68, 2004.

[95] AD Rollett. Abnormal grain growth and texture development. In *Recrystallization and Grain Growth, Pts 1 and 2*, pages 1171–1176, 2005.

[96] D Raabe. Cellular automata in materials science with particular reference to recrystallization simulation. *Ann. Rev. Mat. Res.*, 32:53–76, 2002.

[97] Y Liu, T Baudin, and R Penelle. Simulation of normal grain growth by cellular automata. *Scr. Mater.*, 34(11):1679–1683, 1996.

[98] KGF Janssens. Random grid, three dimensional, space time coupled cellular automata for the simulation of recrystallization and grain growth. *Model. Sim. Mat. Sci. Engr.*, 11(2):157–171, 2003.

[99] KGF Janssens, EA Holm, and SM Foiles. Introducing solute drag in irregular cellular automata modeling of grain growth. In *Recrystallization and Grain Growth, Pts 1 and 2*, pages 1045–1050, 2004.

[100] HL Ding, YZ He, LF Liu, and WJ Ding. Cellular automata simulation of grain growth in three dimensions based on the lowest energy principle. *J. Cryst. Growth*, 293(2):489–497, 2006.

[101] YZ He, HL Ding, LF Liu, and K Shin. Computer simulation of 2d grain growth using a cellular automata model based on the lowest energy principle. *Mat. Sci. Engr. A*, 429(1-2):236–246, 2006.

[102] LQ Chen and W Yang. Computer simulation of the domain dynamics of a quenched system with a large number of nonconserved order parameters - the grain growth kinetics. *Phys. Rev. B*, 50(21):15752–15756, 1994.

[103] LQ Chen. A novel computer simulation technique for modeling grain growth. *Scr. Metal. Mat.*, 32(1):115–120, 1995.

[104] DN Fan, CW Geng, and LQ Chen. Computer simulation of topological evolution in 2-d grain growth using a continuum diffuse interface field model. *Acta Mater.*, 45(3):1115–1126, 1997.

[105] DN Fan and LQ Chen. Diffusion controlled grain growth in two phase solids. *Acta Mater.*, 45(8):3297–3310, 1997.

[106] D Fan and LQ Chen. Computer simulation of grain growth using a continuum field model. *Acta Mater.*, 45(2):611–622, 1997.

[107] D Fan, SP Chen, and LQ Chen. Computer simulation of grain growth kinetics with solute drag. *J. Mat. Res.*, 14(3):1113–1123, 1999.

[108] H Garcke, B Nestler, and B Stoth. A multiphase field concept: Numerical simulations of moving phase boundaries and multiple junctions. *Siam J. Appl. Mathematics*, 60(1):295–315, 1999.

[109] B Nestler. A multiphase-field model: Sharp interface asymptotics and numerical simulations of moving phase boundaries and multijunctions. *J. Cryst. Growth*, 204(1-2):224–228, 1999.

[110] A Kazaryan, Y Wang, SA Dregia, and BR Patton. Generalized phase field model for computer simulation of grain growth in anisotropic systems. *Phys. Rev. B*, 61(21):14275–14278, 2000.

[111] LQ Chen. Phase field models for microstructure evolution. *Ann. Rev. Mat. Res.*, 32:113–140, 2002.

[112] SG Kim, DI Kim, WT Kim, and YB Park. Computer simulations of two dimensional and three dimensional ideal grain growth. *Phys. Rev. E*, 74(6), 2006.

[113] J Gruber, N Ma, Y Wang, AD Rollett, and GS Rohrer. Sparse data structure and algorithm for the phase field method. *Model. Sim. Mat. Sci. Engr.*, 14(7):1189–1195, 2006.

[114] NN Carlson and K Miller. Design and application of a gradient weighted moving finite element code i: in one dimension. *Siam J. Sci. Comp.*, 19(3):728–765, 1998.

[115] NN Carlson and K Miller. Design and application of a gradient weighted moving finite element code ii: in two dimensions. *Siam J. Sci. Comp.*, 19(3):766–798, 1998.

[116] A Kuprat. Modeling microstructure evolution using gradient weighted moving finite elements. *Siam J. Sci. Comp.*, 22(2):535–560, 2000.

[117] A Kuprat, D George, G Straub, and MC Demirel. Modeling microstructure evolution in three dimensions with grain3d and lagrit. *Comp. Mat. Sci.*, 28(2):199–208, 2003.

[118] D Kinderlehrer, J Lee, I Livshits, A Rollett, and S Ta'asan. Mesoscale simulation of grain growth. In *Recrystallization and Grain Growth, Parts 1 and 2*, volume 467-470, pages 1057–1062, 2004.

[119] D Kinderlehrer, I Livshits, and S Ta'asan. A variational approach to modeling and simulation of grain growth. *Siam J. Sci. Comp.*, 28(5):1694–1715, 2006.

[120] M Upmanyu, GN Hassold, A Kazaryan, EA Holm, Y Wang, B Patton, and DJ Srolovitz. Boundary mobility and energy anisotropy effects on microstructural evolution during grain growth. *Interface Sci.*, 10(2-3):201–216, 2002.

[121] GS Grest, DJ Srolovitz, and MP Anderson. Computer simulation of grain growth 4. anisotropic grain boundary energies. *Acta Metall.*, 33(3):509–520, 1985.

[122] Y Saad. *Iterative Methods for Sparse Linear Systems.* PWS Pub. Co., Boston, 1996.

[123] M Hillert. On theory of normal and abnormal grain growth. *Acta Metall.*, 13(3):227, 1965.

[124] F Wakai, Y Shinoda, S Ishihara, and A Dominguez-Rodriguez. Topological transformation of grains in three-dimensional normal grain growth. *J. Mat. Res.*, 16(7):2136–2142, 2001.

[125] MA Fortes and AC Ferro. Topology and transformations in cellular structures. *Acta Metall.*, 33(9):1697–1708, 1985.

[126] CS Smith. *Grain Shapes and Other Applications of Topology*, pages 65–113. American Society of Metals, Cleveland, OH, 1951.

Appendix A

Source code

This appendix contains a listing of all source code developed for use in this work. The first section introduces the Mesoscale Microstructure Simulation Package (MMSP). MMSP is nothing more than a collection of C++ header files that declare a number of grid or mesh objects (classes) and define how most of their methods (member functions) are implemented. We have listed only that source code for MMSP components used in this work; the source for these and other components, as well as documentation, can be found at http://www.matforge.org/cmu/. The next section contains a collection of classes and subroutines for computations involving crystallographic orientations, rotations, and symmetry. The final two sections present those functions used explicitly in our simulations and microstructural analysis.

A.1 MMSP

The following code contains definitions and implementations for the various classes meant to represent the basic data structures at individual grid nodes: **scalar**, **vector**, **sparse**, and **unused**.

```
// MMSP.data.hpp
// Class definitions for MMSP data structures
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef MMSP_DATA
#define MMSP_DATA
#include<map>
#include<cmath>
#include<vector>
#include<complex>
#include<algorithm>

namespace MMSP{

template<typename value_type> class unused{
public:
  // constructors
  explicit unused(...) {}

  // buffer I/O
  int buffer_size() const {return 0;}
  void to_buffer(char* buffer) const {}
  void from_buffer(const char* buffer) {}

  // assignment operators
  unused& operator=(const value_type& temp) {return *this;}
  unused& operator=(const unused& temp) {return *this;}

  // data access operators
  operator value_type&() {return static_cast<value_type>(0);}
  operator const value_type&() const {return static_cast<value_type>(0);}

  // data structure sizes
  void resize(int n) const {}
  int fields() const {return 0;}
  int nonzero() const {return 0;}
};

template<typename value_type> class scalar{
public:
  // constructors
  explicit scalar(...) {}

  // buffer I/O
  int buffer_size() const {return sizeof(data);}
  void to_buffer(char* buffer) const {memcpy(buffer,&data,sizeof(data));}
  void from_buffer(const char* buffer) {memcpy(&data,buffer,sizeof(data));}

  // assignment operators
  scalar& operator=(const value_type& temp) {data=temp; return *this;}
  scalar& operator=(const scalar& temp) {data=temp.data; return *this;}

  // data access operators
  operator value_type&() {return data;}
  operator const value_type&() const {return data;}

  // data structure sizes
  void resize(int n) const {}
  int fields() const {return 1;}
  int nonzero() const {return (data!=static_cast<value_type>(0));}

private:
  value_type data;
};
```

```
template<typename value_type> class vector{
public:
  // constructors
  explicit vector(int n = 1, ...) {data.resize(n);}

  // buffer I/O
  int buffer_size() const {return data.size()*sizeof(value_type);}
  void to_buffer(char* buffer) const {memcpy(buffer,&data[0],data.size()*sizeof(value_type));}
  void from_buffer(const char* buffer) {memcpy(&data[0],buffer,data.size()*sizeof(value_type));}

  // assignment operators
  vector& operator=(const vector& temp) {data=temp.data; return *this;}

  // data access operators
  value_type& operator[](int i) {return data[i];}
  const value_type& operator[](int i) const {return data[i];}

  // data structure sizes
  void resize(int n) {return data.resize(n);}
  int fields() const {return data.size();}
  int nonzero() const {return count_if(data.begin(),data.end(),
                      bind2nd(std::not_equal_to<value_type>(),static_cast<value_type>(0)));}

private:
  std::vector<value_type> data;
};


template<typename value_type> class sparse{
public:
  // constructors
  explicit sparse(...) {}

  // buffer I/O
  int buffer_size() const;
  void to_buffer(char* buffer) const;
  void from_buffer(const char* buffer);

  // assignment operators
  sparse& operator=(const sparse& temp) {data=temp.data; return *this;}

  // data access operators
  value_type& operator[](int i);
  const value_type operator[](int i) const;

  // data structure sizes
  void resize(int n) const {}
  int fields() const {return data.size();}
  int nonzero() const {return data.size();}

  // utility functions
  int index(int i) const {return data[i].first;}
  value_type value(int i) const {return data[i].second;}

private:
  std::vector<std::pair<int,value_type> > data;
};

template <typename value_type>
value_type& sparse<value_type>::operator[](int index)
{
  int n = data.size();
  for (int i=0; i<n; i++)
    if (data[i].first==index) return data[i].second;
  data.push_back(std::make_pair(index,static_cast<value_type>(0)));
  return data.back().second;
}

template <typename value_type>
const value_type sparse<value_type>::operator[](int index) const
{
  int n = data.size();
  for (int i=0; i<n; i++)
    if (data[i].first==index) return data[i].second;
  return static_cast<value_type>(0);
}

template <typename value_type>
int sparse<value_type>::buffer_size() const
{
  int n = data.size();
  return sizeof(int)+sizeof(std::pair<int,value_type>);
}

template <typename value_type>
void sparse<value_type>::to_buffer(char* buffer) const
{
  char* p = buffer;
  int n = data.size();
  memcpy(p,&n,sizeof(int));
  p += sizeof(int);
  memcpy(p,&data[0],n*sizeof(std::pair<int,value_type>));
}

template <typename value_type>
void sparse<value_type>::from_buffer(const char* buffer)
{
  const char* p = buffer;
  int n;
  memcpy(&n,p,sizeof(int));
  p += sizeof(int);
  data.resize(n);
  memcpy(&data[0],p,n*sizeof(std::pair<int,value_type>));
}

} // namespace MMSP

#endif
```

This file contains definitions and serial implementations for the basic **grid** template class, as well as for the intermediate **grid1D**, **grid2D**, and **grid3D** classes, used mainly in deriving the high level grid classes associated with various computational methods.

```
// MMSP.grid.hpp
// Class definitions for rectilinear grids (base class)
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef MMSP_GRID
#define MMSP_GRID
#include"MMSP.data.hpp"
#include<iostream>
#include<fstream>
#include<cstdarg>

namespace MMSP{

template <int dimension, typename type>
class grid{
public:
  // constructors
  grid(int nf, ...);
  grid(const char* filename) {this->input(filename);}
  grid(const char* filename, int id, int np, int ng = 1);

  // subscript operator
  type& operator[](int i) {return data[i];}
  const type& operator[](int i) const {return data[i];}

  // buffer I/O
  int buffer_size(int x0 = 0, int sx = 0) const;
  void to_buffer(char* buffer, int x0 = 0, int sx = 0) const;
  void from_buffer(char* buffer, int x0 = 0, int sx = 0);

  // file I/O
  void input(const char* filename);
  void input(const char* filename, int id, int np, int ng = 1);
  void output(const char* filename) const;
  void output(const char* filename, int id, int np, int ng = 1) const;

  // grid attribute functions
  int fields() const {return data[0].fields();}
  int size(int i) const {return nx[i];}
  bool boundary(int i) const {return px[i];}
  bool& boundary(int i) {return px[i];}
  float spacing(int i) const {return dx[i];}
  float& spacing(int i) {return dx[i];}

  // grid operations
  void swap(grid& temp) {data.swap(temp.data);}
  void ghostswap(int id, int np, int ng = 1);

protected:
  // grid data
  std::vector<type> data;
  int   nx[dimension];
  bool  px[dimension];
  float dx[dimension];
};

template <int dimension, typename type>
grid<dimension,type>::grid(int nf, ...)
{
  va_list list;
  va_start(list,nf);
  for (int i=0; i<dimension; i++) {
    nx[i] = va_arg(list,int);
    px[i] = false;
    dx[i] = 1.0;
  }
  va_end(list);

  if (dimension==1) data.resize(nx[0],type(nf));
  if (dimension==2) data.resize(nx[0],type(nf,nx[1]));
  if (dimension==3) data.resize(nx[0],type(nf,nx[1],nx[2]));
}

template <int dimension, typename type>
int grid<dimension,type>::buffer_size(int x0, int sx) const
{
  int buffer_size = 0;
  if (sx==0) sx = data.size();
  for (int x=x0; x<x0+sx; x++)
    buffer_size += data[x].buffer_size();
  return buffer_size;
}

template <int dimension, typename type>
void grid<dimension,type>::to_buffer(char* buffer, int x0, int sx) const
{
  char* p = buffer;
  if (sx==0) sx = data.size();
  for (int x=x0; x<x0+sx; x++) {
    data[x].to_buffer(p);
    p += data[x].buffer_size();
  }
}

template <int dimension, typename type>
void grid<dimension,type>::from_buffer(char* buffer, int x0, int sx)
{
  char* p = buffer;
  if (sx==0) sx = data.size();
  for (int x=x0; x<x0+sx; x++) {
    data[x].from_buffer(p);
    p += data[x].buffer_size();
  }
}

template <int dimension, typename type>
void grid<dimension,type>::input(const char* filename)
{
  // file open error check
  std::ifstream input(filename);
  if (!input) {
    std::cerr<<"file input error: "<<filename;
    std::cerr<<" cannot be opened."<<std::endl;
    exit(-1);
  }

  // dimension error check
  int dim;
  input.read(reinterpret_cast<char*>(&dim),sizeof(dim));
  if (dim!=dimension) {
    std::cerr<<"File input error: "<<filename;
    std::cerr<<" has grid of dimension "<<dim<<std::endl;
    exit(-2);
  }

  // read number of fields
  int nf;
  input.read(reinterpret_cast<char*>(&nf),sizeof(nf));

  // read grid parameters
  for (int i=0; i<dimension; i++)
    input.read(reinterpret_cast<char*>(&nx[i]),sizeof(nx[i]));
  for (int i=0; i<dimension; i++)
    input.read(reinterpret_cast<char*>(&px[i]),sizeof(px[i]));
  for (int i=0; i<dimension; i++)
    input.read(reinterpret_cast<char*>(&dx[i]),sizeof(dx[i]));

  // resize grid data structure
  if (dimension==1) data.resize(nx[0],type(nf));
  if (dimension==2) data.resize(nx[0],type(nf,nx[1]));
  if (dimension==3) data.resize(nx[0],type(nf,nx[1],nx[2]));

  // read grid data
  int pos1 = input.tellg();
  input.seekg(0,std::ios::end);
  int pos2 = input.tellg();
  input.seekg(pos1,std::ios::beg);
  int size = pos2-pos1;
  char* buffer = new char[size];
  input.read(reinterpret_cast<char*>(buffer),size);
  this->from_buffer(buffer);
  delete [] buffer;

  // seed random number generators
  srand(time(NULL));
}

template <int dimension, typename type>
void grid<dimension,type>::output(const char* filename) const
{
  // file open error check
  std::ofstream output(filename);
  if (!output) {
    std::cerr<<"file input error: "<<filename;
    std::cerr<<" cannot be opened."<<std::endl;
    exit(-1);
  }

  // write grid dimension
  int dim = dimension;
  output.write(reinterpret_cast<const char*>(&dim),sizeof(dim));

  // write number of fields
  int nf = fields();
  output.write(reinterpret_cast<const char*>(&nf),sizeof(nf));

  // write grid parameters
  for (int i=0; i<dimension; i++)
    output.write(reinterpret_cast<const char*>(&nx[i]),sizeof(nx[i]));
  for (int i=0; i<dimension; i++)
    output.write(reinterpret_cast<const char*>(&px[i]),sizeof(px[i]));
  for (int i=0; i<dimension; i++)
    output.write(reinterpret_cast<const char*>(&dx[i]),sizeof(dx[i]));

  // write grid data
  int size = this->buffer_size();
  char* buffer = new char[size];
  this->to_buffer(buffer);
  output.write(reinterpret_cast<const char*>(buffer),size);
  delete [] buffer;
}


template < template<typename value_type> class data_type, typename value_type >
class grid1D : public grid<1,data_type<value_type> > {
public:
  // constructors
  grid1D(int x, int nf = 1)
    : grid<1,data_type<value_type> > (nf,x) {}
  grid1D(const char* filename)
    : grid<1,data_type<value_type> > (filename) {}
  grid1D(const char* filename, int id, int np, int ng = 1)
    : grid<1,data_type<value_type> > (filename,id,np,ng) {}

  // utility functions
  const data_type<value_type>& neighbor(int x, int sx) const;
};

template < template<typename value_type> class data_type, typename value_type >
const data_type<value_type>& grid1D<data_type,value_type>::
  neighbor(int x, int sx) const
{
  const grid1D& grid = *this;
  int  nx = grid.nx[0];
  bool px = grid.px[0];

  int xsx = x+sx;
  int ax = (xsx>=nx);
  int bx = (xsx<0);
  int i = xsx+(px)*(nx*(bx-ax))+(!px)*(ax*(nx-1)-(ax|bx)*xsx);

  return grid.data[i];
}


template < template<typename value_type> class data_type, typename value_type >
class grid2D : public grid<2,grid<1,data_type<value_type> > > {
public:
  // constructors
  grid2D(int x, int y, int nf = 1)
    : grid<2,grid<1,data_type<value_type> > > (nf,x,y) {}
  grid2D(const char* filename)
    : grid<2,grid<1,data_type<value_type> > > (filename) {}
  grid2D(const char* filename, int id, int np, int ng = 1)
    : grid<2,grid<1,data_type<value_type> > > (filename,id,np,ng) {}

  // utility functions
  const data_type<value_type>& neighbor(int x, int y, int sx, int sy) const;
};

template < template<typename value_type> class data_type, typename value_type >
const data_type<value_type>& grid2D<data_type,value_type>::
  neighbor(int x, int y, int sx, int sy) const
{
  const grid2D& grid = *this;
  int  nx = grid.nx[0];
  int  ny = grid.nx[1];
  bool px = grid.px[0];
  bool py = grid.px[1];

  int xsx = x+sx;
  int ax = (xsx>=nx);
  int bx = (xsx<0);
  int i = xsx+(px)*(nx*(bx-ax))+(!px)*(ax*(nx-1)-(ax|bx)*xsx);

  int ysy = y+sy;
  int ay = (ysy>=ny);
  int by = (ysy<0);
  int j = ysy+(py)*(ny*(by-ay))+(!py)*(ay*(ny-1)-(ay|by)*ysy);

  return grid.data[i][j];
}


template < template<typename value_type> class data_type, typename value_type >
class grid3D : public grid<3,grid<2,grid<1,data_type<value_type> > > > {
public:
  // constructors
  grid3D(int x, int y, int z, int nf = 1)
    : grid<3,grid<2,grid<1,data_type<value_type> > > > (nf,x,y,z) {}
  grid3D(const char* filename)
    : grid<3,grid<2,grid<1,data_type<value_type> > > > (filename) {}
  grid3D(const char* filename, int id, int np, int ng = 1)
    : grid<3,grid<2,grid<1,data_type<value_type> > > > (filename,id,np,ng) {}

  // utility functions
  const data_type<value_type>& neighbor(int x, int y, int z, int sx, int sy, int sz) const;
};

template < template<typename value_type> class data_type, typename value_type >
const data_type<value_type>& grid3D<data_type,value_type>::
  neighbor(int x, int y, int z, int sx, int sy, int sz) const
```

```
{
  const grid3D& grid = *this;
  int  nx = grid.nx[0];
  int  ny = grid.nx[1];
  int  nz = grid.nx[2];
  bool px = grid.px[0];
  bool py = grid.px[1];
  bool pz = grid.px[2];

  int xsx = x+sx;
  int ax = (xsx>=nx);
  int bx = (xsx<0);
  int i = xsx+(px)*(nx*(bx-ax))+(!px)*(ax*(nx-1)-(ax|bx)*xsx);

  int ysy = y+sy;
  int ay = (ysy>=ny);
  int by = (ysy<0);
  int j = ysy+(py)*(ny*(by-ay))+(!py)*(ay*(ny-1)-(ay|by)*ysy);

  int zsz = z+sz;
  int az = (zsz>=nz);
  int bz = (zsz<0);
  int k = zsz+(pz)*(nz*(bz-az))+(!pz)*(az*(nz-1)-(az|bz)*zsz);

  return grid.data[i][j][k];
}

} // namespace MMSP

#endif
```

This file contains the parallel implementations for the basic MMSP **grid** template class' parallel I/O and ghost cell swapping functions. The separation of serial and parallel implementations is motivated by the expected need to compile serial code without using the MPI libraries.

```
// MMSP.grid.parallel.hpp
// Implementation of parallel grid operations (requires MPI)
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef MMSP_GRID_PARALLEL
#define MMSP_GRID_PARALLEL
#include"MMSP.grid.hpp"
#include"mpicxx.h"

namespace MMSP{

template <int dimension, typename type>
grid<dimension,type>::grid(const char* filename, int id, int np, int ng)
{
  this->input(filename,id,np,ng);
}

template <int dimension, typename type>
void grid<dimension,type>::input(const char* filename, int id, int np, int ng)
{
  if (id==0) {
    // file open error check
    std::ifstream input(filename);
    if (!input) {
      std::cerr<<"file input error: "<<filename;
      std::cerr<<" cannot be opened."<<std::endl;
      exit(-1);
    }

    // dimension error check
    int dim;
    input.read(reinterpret_cast<char*>(&dim),sizeof(dim));
    if (dim!=dimension) {
      std::cerr<<"File input error: "<<filename;
      std::cerr<<" has grid of dimension "<<dim<<std::endl;
      exit(-2);
    }

    // read number of fields
    int gnf;
    input.read(reinterpret_cast<char*>(&gnf),sizeof(gnf));

    // read global grid parameters
    int   gnx[dimension];
    bool  gpx[dimension];
    float gdx[dimension];
    for (int i=0; i<dimension; i++)
      input.read(reinterpret_cast<char*>(&gnx[i]),sizeof(nx[i]));
    for (int i=0; i<dimension; i++)
      input.read(reinterpret_cast<char*>(&gpx[i]),sizeof(px[i]));
    for (int i=0; i<dimension; i++)
      input.read(reinterpret_cast<char*>(&gdx[i]),sizeof(dx[i]));

    // read global grid data
    int pos1 = input.tellg();
    input.seekg(0,std::ios::end);
    int pos2 = input.tellg();
    input.seekg(pos1,std::ios::beg);
    int size = pos2-pos1;
    char* buffer = new char[size];
    input.read(reinterpret_cast<char*>(buffer),size);

    // set main buffer position
    char* p = buffer;

    for (int ip=0; ip<np; ip++) {
      // compute local grid parameters
      int lnx[dimension];
       lnx[0] = (ip!=np-1)*(gnx[0]/np)+(ip==np-1)*(gnx[0]-(gnx[0]/np)*(np-1))+2*ng;
      for (int i=1; i<dimension; i++) lnx[i] = gnx[i];

      // send local grid parameters
      MPI::COMM_WORLD.Send(&gnf,1,MPI_INT,ip,100);
      for (int i=0; i<dimension; i++)
        MPI::COMM_WORLD.Send(&lnx[i],1,MPI_INT,ip,110+i);
      for (int i=0; i<dimension; i++)
        MPI::COMM_WORLD.Send(&gpx[i],1,MPI_CHAR,ip,120+i);
      for (int i=0; i<dimension; i++)
        MPI::COMM_WORLD.Send(&gdx[i],1,MPI_FLOAT,ip,130+i);

      // read local grid data
      grid temp(gnf,lnx[0],lnx[1],lnx[2]);
      temp.from_buffer(p,ng,lnx[0]-2*ng);

      // send local grid data
      int subgrid_size = temp.buffer_size(ng,lnx[0]-2*ng);
      char* subgrid_buffer = new char[subgrid_size];
      temp.to_buffer(subgrid_buffer,ng,lnx[0]-2*ng);
      MPI::COMM_WORLD.Send(&subgrid_size,1,MPI_INT,ip,200);
      MPI::COMM_WORLD.Send(subgrid_buffer,subgrid_size,MPI_CHAR,ip,300);
      delete [] subgrid_buffer;

      // advance buffer position
      p += subgrid_size;
    }

    delete [] buffer;
    input.close();
  }

  // receive grid parameters
  int nf;
  MPI::COMM_WORLD.Recv(&nf,1,MPI_INT,0,100);
  for (int i=0; i<dimension; i++)
    MPI::COMM_WORLD.Recv(&nx[i],1,MPI_INT,0,110+i);
  for (int i=0; i<dimension; i++)
    MPI::COMM_WORLD.Recv(&px[i],1,MPI_CHAR,0,120+i);
  for (int i=0; i<dimension; i++)
    MPI::COMM_WORLD.Recv(&dx[i],1,MPI_FLOAT,0,130+i);

  // resize grid data structure
  if (dimension==1) data.resize(nx[0],type(nf));
  if (dimension==2) data.resize(nx[0],type(nf,nx[1]));
  if (dimension==3) data.resize(nx[0],type(nf,nx[1],nx[2]));

  // receive grid data
  int size;
  MPI::COMM_WORLD.Recv(&size,1,MPI_INT,0,200);
  char* buffer = new char[size];
  MPI::COMM_WORLD.Recv(buffer,size,MPI_CHAR,0,300);
  this->from_buffer(buffer,ng,nx[0]-2*ng);
  delete [] buffer;

  // send and receive ghost cells
  this->ghostswap(id,np,ng);

  // seed random number generators
  srand(time(NULL));
}

template <int dimension, typename type>
void grid<dimension,type>::output(const char* filename, int id, int np, int ng) const
{
  // send local grid parameters
  MPI::COMM_WORLD.Send(&nx[0],1,MPI_INT,0,100);

  // send grid data
  int size = this->buffer_size(ng,nx[0]-2*ng);
  MPI::COMM_WORLD.Send(&size,1,MPI_INT,0,200);
  char* buffer = new char[size];
  this->to_buffer(buffer,ng,nx[0]-2*ng);
  MPI::COMM_WORLD.Send(buffer,size,MPI_CHAR,0,300);
  delete [] buffer;

  if (id==0) {
    // file open error check
    std::ofstream output(filename);
    if (!output) {
      std::cerr<<"file input error: "<<filename;
      std::cerr<<" cannot be opened."<<std::endl;
      exit(-1);
    }

    // write grid dimension
    int dim = dimension;
    output.write(reinterpret_cast<const char*>(&dim),sizeof(dim));

    // write number of fields
    int nf = fields();
    output.write(reinterpret_cast<const char*>(&nf),sizeof(nf));

    // write grid parameters
    for (int i=0; i<dimension; i++)
      output.write(reinterpret_cast<const char*>(&nx[i]),sizeof(nx[i]));
    for (int i=0; i<dimension; i++)
      output.write(reinterpret_cast<const char*>(&px[i]),sizeof(px[i]));
    for (int i=0; i<dimension; i++)
      output.write(reinterpret_cast<const char*>(&dx[i]),sizeof(dx[i]));

    // global nx is wrong
    int gnx = 0;

    for (int ip=0; ip<np; ip++) {
      // receive local grid parameters
      int lnx;
      MPI::COMM_WORLD.Recv(&lnx,1,MPI_INT,ip,100);
      gnx += lnx-2*ng;

      // receive local grid data
      int subgrid_size;
      MPI::COMM_WORLD.Recv(&subgrid_size,1,MPI_INT,ip,200);
      char* subgrid_buffer = new char[subgrid_size];
      MPI::COMM_WORLD.Recv(subgrid_buffer,subgrid_size,MPI_CHAR,ip,300);

      // write local grid data
      output.write(reinterpret_cast<const char*>(subgrid_buffer),subgrid_size);
      delete [] subgrid_buffer;
    }

    // correct global nx
    output.seekp(sizeof(dim)+sizeof(nf));
    output.write(reinterpret_cast<const char*>(&gnx),sizeof(gnx));
    output.close();
  }
}

template <int dimension, typename type>
void grid<dimension,type>::ghostswap(int id, int np, int ng)
{
  // swap ghost cells with other processes
  // global domain must be subdivided into slabs

  // declare buffers and sizes
  char* send_buffer;
  char* recv_buffer;
  int send_size;
  int recv_size;

  // send to processor above and receive from processor below
  send_size = this->buffer_size(nx[0]-2*ng,ng);
  MPI::COMM_WORLD.Send(&send_size,1,MPI_INT,(id+np+1)%np,100);
  MPI::COMM_WORLD.Recv(&recv_size,1,MPI_INT,(id+np-1)%np,100);
  send_buffer = new char[send_size];
  recv_buffer = new char[recv_size];
  this->to_buffer(send_buffer,nx[0]-2*ng,ng);
  MPI::COMM_WORLD.Issend(send_buffer,send_size,MPI_CHAR,(id+np+1)%np,200);
  MPI::COMM_WORLD.Recv(recv_buffer,recv_size,MPI_CHAR,(id+np-1)%np,200);
  MPI::COMM_WORLD.Barrier();
  this->from_buffer(recv_buffer,0,ng);
  delete [] send_buffer;
  delete [] recv_buffer;

  // send to processor below and receive from processor above
  send_size = this->buffer_size(ng,ng);
  MPI::COMM_WORLD.Send(&send_size,1,MPI_INT,(id+np-1)%np,100);
  MPI::COMM_WORLD.Recv(&recv_size,1,MPI_INT,(id+np+1)%np,100);
  send_buffer = new char[send_size];
  recv_buffer = new char[recv_size];
  this->to_buffer(send_buffer,ng,ng);
  MPI::COMM_WORLD.Issend(send_buffer,send_size,MPI_CHAR,(id+np-1)%np,200);
  MPI::COMM_WORLD.Recv(recv_buffer,recv_size,MPI_CHAR,(id+np+1)%np,200);
  MPI::COMM_WORLD.Barrier();
  this->from_buffer(recv_buffer,nx[0]-ng,ng);
  delete [] send_buffer;
  delete [] recv_buffer;

  // correct for non-periodic boundary conditions
  if (px[0]==false) {
    if (id==0) {
      int size = this->buffer_size(ng,1);
      char* buffer = new char[size];
      this->to_buffer(buffer,ng,1);
      for (int x=0; x<ng; x++)
        this->from_buffer(buffer,x,1);
      delete [] buffer;
    }

    if (id==np-1) {
      int size = this->buffer_size(nx[0]-ng-1,1);
      char* buffer = new char[size];
      this->to_buffer(buffer,nx[0]-ng-1,1);
      for (int x=nx[0]-ng; x<nx[0]; x++)
        this->from_buffer(buffer,x,1);
      delete [] buffer;
    }
  }
}

} // namespace MMSP

#endif
```

This code contains the definition and implementation of all high level grids used for Monte Carlo simulations.

```
// MCgrid.hpp
// Class definitions for rectilinear Monte Carlo grids
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef MCGRID
#define MCGRID
#include"MMSP.grid.hpp"

namespace MMSP{

class MCgrid1D : public grid1D<scalar,int> {
public:
  // constructors
  MCgrid1D(int x)
    : grid1D<scalar,int>(x) {}
  MCgrid1D(const char* filename)
    : grid1D<scalar,int>(filename) {}
  MCgrid1D(const char* filename, int id, int np, int ng = 1)
    : grid1D<scalar,int>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // utility functions
  std::vector<int> nonzero(int x) const;
};

std::vector<int> MCgrid1D::nonzero(int x) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    //if (abs(i)<=1)
      nonzero.push_back(neighbor(x,i));
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}


class MCgrid2D : public grid2D<scalar,int> {
public:
  // constructors
  MCgrid2D(int x, int y)
    : grid2D<scalar,int>(x,y) {}
  MCgrid2D(const char* filename)
    : grid2D<scalar,int>(filename) {}
  MCgrid2D(const char* filename, int id, int np, int ng = 1)
    : grid2D<scalar,int>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // utility functions
  std::vector<int> nonzero(int x, int y) const;
};

std::vector<int> MCgrid2D::nonzero(int x, int y) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    for (int j=-1; j<=1; j++)
      //if (abs(i)+abs(j)<=1)
        nonzero.push_back(neighbor(x,y,i,j));
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}


class MCgrid3D : public grid3D<scalar,int> {
public:
  // constructors
  MCgrid3D(int x, int y, int z)
    : grid3D<scalar,int>(x,y,z) {}
  MCgrid3D(const char* filename)
    : grid3D<scalar,int>(filename) {}
  MCgrid3D(const char* filename, int id, int np, int ng = 1)
    : grid3D<scalar,int>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // utility functions
  std::vector<int> nonzero(int x, int y, int z) const;
};

std::vector<int> MCgrid3D::nonzero(int x, int y, int z) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    for (int j=-1; j<=1; j++)
      for (int k=-1; k<=1; k++)
        //if (abs(i)+abs(j)+abs(k)<=1)
          nonzero.push_back(neighbor(x,y,z,i,j,k));
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}

} // namespace MMSP

#endif
```

This code contains the definition and implementation of all high level grids used for phase field simulations using the sparsePF data structure.

```
// sparsePF.hpp
// Class definitions for rectilinear sparsePF method grids
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef sparsePFGRID
#define sparsePFGRID
#include"MMSP.grid.hpp"

namespace MMSP{

class sparsePF1D : public grid1D<sparse,float> {
public:
  // constructors
  sparsePF1D(int x)
    : grid1D<sparse,float>(x) {}
  sparsePF1D(const char* filename)
    : grid1D<sparse,float>(filename) {}
  sparsePF1D(const char* filename, int id, int np, int ng = 1)
    : grid1D<sparse,float>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // numerical functions
  float laplacian(int x, int index) const;

  // utility functions
  std::vector<int> nonzero(int x) const;
};

float sparsePF1D::laplacian(int x, int index) const
{
  return (neighbor(x,1)[index]+neighbor(x,-1)[index]
         -2.0*neighbor(x,0)[index])/(dx[0]*dx[0]);
}

std::vector<int> sparsePF1D::nonzero(int x) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    if (abs(i)<=1) {
      const sparse<float>& data = neighbor(x,i);
      for (int h=0; h<data.fields(); h++)
        nonzero.push_back(data.index(h));
    }
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}


class sparsePF2D : public grid2D<sparse,float> {
public:
  // constructors
  sparsePF2D(int x, int y)
    : grid2D<sparse,float>(x,y) {}
  sparsePF2D(const char* filename)
    : grid2D<sparse,float>(filename) {}
  sparsePF2D(const char* filename, int id, int np, int ng = 1)
    : grid2D<sparse,float>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // numerical functions
  float laplacian(int x, int y, int index) const;

  // utility functions
  std::vector<int> nonzero(int x, int y) const;
};

float sparsePF2D::laplacian(int x, int y, int index) const
{
  return (neighbor(x,y,1,0)[index]+neighbor(x,y,-1,0)[index]
         -2.0*neighbor(x,y,0,0)[index])/(dx[0]*dx[0])
        +(neighbor(x,y,0,1)[index]+neighbor(x,y,0,-1)[index]
         -2.0*neighbor(x,y,0,0)[index])/(dx[1]*dx[1]);
}

std::vector<int> sparsePF2D::nonzero(int x, int y) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    for (int j=-1; j<=1; j++)
      if (abs(i)+abs(j)<=1) {
        const sparse<float>& data = neighbor(x,y,i,j);
        for (int h=0; h<data.fields(); h++)
          nonzero.push_back(data.index(h));
      }
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}


class sparsePF3D : public grid3D<sparse,float> {
public:
  // constructors
  sparsePF3D(int x, int y, int z)
    : grid3D<sparse,float>(x,y,z) {}
  sparsePF3D(const char* filename)
    : grid3D<sparse,float>(filename) {}
  sparsePF3D(const char* filename, int id, int np, int ng = 1)
    : grid3D<sparse,float>(filename,id,np,ng) {}

  // the update function
  void update(int steps = 1);
  void update(int steps, int id, int np, int ng = 1);

  // numerical functions
  float laplacian(int x, int y, int z, int index) const;

  // utility functions
  std::vector<int> nonzero(int x, int y, int z) const;
};

float sparsePF3D::laplacian(int x, int y, int z, int index) const
{
  return (neighbor(x,y,z,1,0,0)[index]+neighbor(x,y,z,-1,0,0)[index]
         -2.0*neighbor(x,y,z,0,0,0)[index])/(dx[0]*dx[0])
        +(neighbor(x,y,z,0,1,0)[index]+neighbor(x,y,z,0,-1,0)[index]
         -2.0*neighbor(x,y,z,0,0,0)[index])/(dx[1]*dx[1])
        +(neighbor(x,y,z,0,0,1)[index]+neighbor(x,y,z,0,0,-1)[index]
         -2.0*neighbor(x,y,z,0,0,0)[index])/(dx[2]*dx[2]);
}

std::vector<int> sparsePF3D::nonzero(int x, int y, int z) const
{
  std::vector<int> nonzero;
  for (int i=-1; i<=1; i++)
    for (int j=-1; j<=1; j++)
      for (int k=-1; k<=1; k++)
        if (abs(i)+abs(j)+abs(k)<=1) {
          const sparse<float>& data = neighbor(x,y,z,i,j,k);
          for (int h=0; h<data.fields(); h++)
            nonzero.push_back(data.index(h));
        }
  sort(nonzero.begin(),nonzero.end());
  nonzero.erase(unique(nonzero.begin(),nonzero.end()),nonzero.end());
  return nonzero;
}

} // namespace MMSP

#endif
```

A.2 Texture

This file contains the quaternion class definition and implementation. Only generic quaternion operations are defined here; functions that interpret quaternions as rotation operators are defined elsewhere. Note that quaternion components use the order given by $q_0 + q_1 i + q_2 j + q_3 k$.

```
// quaternion.hpp
// Quaternion class definition and implementation
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef QUATERNION
#define QUATERNION

template<typename T> class quaternion;

template<typename T>
inline quaternion<T> conj(const quaternion<T>& q)
{
  return quaternion<T>(q[0],-q[1],-q[2],-q[3]);
}

template<typename T>
inline T norm(const quaternion<T>& q)
{
  return sqrt(q[0]*q[0]+q[1]*q[1]+q[2]*q[2]+q[3]*q[3]);
}

template<typename T>
inline bool operator==(const quaternion<T>& q, const quaternion<T>& p)
{
  return (q[0]==p[0] && q[1]==p[1] && q[2]==p[2] && q[3]==p[3]);
}

template<typename T>
inline bool operator==(const T& c, const quaternion<T>& q)
{
  return (quaternion<T>(c)==q);
}

template<typename T>
inline bool operator==(const quaternion<T>& q, const T& c)
{
  return (q==quaternion<T>(c));
}

template<typename T>
inline bool operator!=(const quaternion<T>& q, const quaternion<T>& p)
{
  return (q[0]!=p[0] || q[1]!=p[1] || q[2]!=p[2] || q[3]!=p[3]);
}

template<typename T>
inline bool operator!=(const T& c, const quaternion<T>& q)
{
  return (quaternion<T>(c)!=q);
}

template<typename T>
inline bool operator!=(const quaternion<T>& q, const T& c)
{
  return (q!=quaternion<T>(c));
}

template<typename T>
inline quaternion<T> operator+(const quaternion<T>& q)
{
  return quaternion<T>(+q[0],+q[1],+q[2],+q[3]);
}
```

```
template<typename T>
inline quaternion<T> operator+(const quaternion<T>& q, const quaternion<T>& p)
{
  return quaternion<T>(q[0]+p[0],q[1]+p[1],q[2]+p[2],q[3]+p[3]);
}

template<typename T>
inline quaternion<T> operator+(const quaternion<T>& q, const T& c)
{
  return q+quaternion<T>(c);
}

template<typename T>
inline quaternion<T> operator+(const T& c, const quaternion<T>& q)
{
  return quaternion<T>(c)+q;
}

template<typename T>
inline quaternion<T> operator-(const quaternion<T>& q)
{
  return quaternion<T>(-q[0],-q[1],-q[2],-q[3]);
}

template<typename T>
inline quaternion<T> operator-(const quaternion<T>& q, const quaternion<T>& p)
{
  return quaternion<T>(q[0]-p[0],q[1]-p[1],q[2]-p[2],q[3]-p[3]);
}

template<typename T>
inline quaternion<T> operator-(const quaternion<T>& q, const T& c)
{
  return q-quaternion<T>(c);
}

template<typename T>
inline quaternion<T> operator-(const T& c, const quaternion<T>& q)
{
  return quaternion<T>(c)-q;
}

template<typename T>
inline quaternion<T> operator*(const quaternion<T>& q, const quaternion<T>& p)
{
  return quaternion<T>(q[0]*p[0]-q[1]*p[1]-q[2]*p[2]-q[3]*p[3],
                       q[0]*p[1]+q[1]*p[0]+q[2]*p[3]-q[3]*p[2],
                       q[0]*p[2]+q[2]*p[0]-q[1]*p[3]+q[3]*p[1],
                       q[0]*p[3]+q[3]*p[0]+q[1]*p[2]-q[2]*p[1]);
}

template<typename T>
inline quaternion<T> operator*(const quaternion<T>& q, const T& c)
{
  return q*quaternion<T>(c);
}

template<typename T>
inline quaternion<T> operator*(const T& c, const quaternion<T>& q)
{
  return quaternion<T>(c)*q;
}

template<typename T>
inline quaternion<T> operator/(const quaternion<T>& q, const quaternion<T>& p)
{
  return q*((1.0/norm(p))*conj(p));
}

template<typename T>
inline quaternion<T> operator/(const quaternion<T>& q, const T& c)
{
  return q/quaternion<T>(c);
}

template<typename T>
inline quaternion<T> operator/(const T& c, const quaternion<T>& q)
{
  return quaternion<T>(c)/q;
}

template<typename T> class quaternion{
public:
  quaternion(T q0=0.0, T q1=0.0, T q2=0.0, T q3=0.0)
    {q[0]=q0; q[1]=q1; q[2]=q2; q[3]=q3;}
  T& operator[](int i)
    {return q[i];}
  const T& operator[](int i) const
    {return q[i];}
  template<typename U> quaternion(const U& c)
    {q[0]=c; q[1]=0.0; q[2]=0.0; q[3]=0.0;}
  template<typename U> quaternion(const quaternion<U>& p)
    {q[0]=p[0]; q[1]=p[1]; q[2]=p[2]; q[3]=p[3];}
  template<typename U> quaternion& operator+=(const U& c)
    {(*this)=(*this)+quaternion(c); return (*this);}
  template<typename U> quaternion& operator-=(const U& c)
    {(*this)=(*this)-quaternion(c); return (*this);}
  template<typename U> quaternion& operator*=(const U& c)
    {(*this)=(*this)*quaternion(c); return (*this);}
  template<typename U> quaternion& operator/=(const U& c)
    {(*this)=(*this)/quaternion(c); return (*this);}
  template<typename U> quaternion& operator+=(const quaternion& p)
    {(*this)=(*this)+quaternion(p); return (*this);}
  template<typename U> quaternion& operator-=(const quaternion& p)
    {(*this)=(*this)-quaternion(p); return (*this);}
  template<typename U> quaternion& operator*=(const quaternion& p)
    {(*this)=(*this)*quaternion(p); return (*this);}
  template<typename U> quaternion& operator/=(const quaternion& p)
    {(*this)=(*this)/quaternion(p); return (*this);}
private:
  T q[4];
};

#endif
```

This file contains useful functions when interpreting quaternions as 3D rotations. Included are functions for conversion to and from a number of other mathematical representations for rotations, as well as a function for computing random quaternion orientations.

```
// rotations.hpp
// Functions for use with unit quaternions (interpreted as 3D rotations)
// Note: an expression such as q1*q2 implies that rotation q2 follows rotation q1
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef ROTATIONS
#define ROTATIONS
#include<cmath>
#include"quaternion.hpp"

template<typename T> quaternion<T> rotation(const T& theta, const T& x, const T& y, const T& z)
{
  T norm = sqrt(x*x+y*y+z*z);
  if (norm>0.0) {
    T rnorm = 1.0/norm;
    T v0 = cos(0.5*theta);
    T v1 = sin(0.5*theta)*rnorm;
    return quaternion<T>(v0,x*v1,y*v1,z*v1);
  }
  return quaternion<T>(1.0);
}

template<typename T> void rotation(const quaternion<T>& q, T& theta, T& x, T& y, T& z)
{
  x = q[1];
  y = q[2];
  z = q[3];
  T norm = sqrt(x*x+y*y+z*z);
  if (norm>0.0) {
    T rnorm = 1.0/norm;
    x *= rnorm;
    y *= rnorm;
    z *= rnorm;
  }
  theta = 2.0*acos(q[0]);
}

template<typename T> quaternion<T> euler(const T& phi1, const T& PHI, const T& phi2)
{
  T hP  = 0.5*PHI;
  T shP = sin(hP);
  T chP = cos(hP);
  T hp1 = 0.5*phi1;
  T hp2 = 0.5*phi2;
  T sum = hp1+hp2;
  T dif = hp1-hp2;

  return quaternion<T>(chP*cos(sum),shP*cos(dif),shP*sin(dif),chP*sin(sum));
}

template<typename T> void euler(const quaternion<T>& q, T& phi1, T& PHI, T& phi2)
{
  T v0  = sqrt(q[0]*q[0]+q[3]*q[3]);
  T v1  = (v0>1.0)? 1.0 : v0;
  T sum = atan2(q[3],q[0]);
  T dif = atan2(q[2],q[1]);

  PHI   = 2.0*acos(v1);
  phi1  = sum+dif;
  phi1 += 2.0*M_PI*(phi1<0.0);
  phi2  = sum-dif;
  phi2 += 2.0*M_PI*(phi2<0.0);
}

template<typename T> quaternion<T> qrandom()
{
  T v0 =          static_cast<T>(rand())/(static_cast<T>(RAND_MAX)+1);
  T v1 = 2.0*M_PI*static_cast<T>(rand())/(static_cast<T>(RAND_MAX)+1);
  T v2 = 2.0*M_PI*static_cast<T>(rand())/(static_cast<T>(RAND_MAX)+1);
  T v3 = sqrt(v0);
  T v4 = sqrt(1.0-v0);

  return quaternion<T>(cos(v1)*v3,sin(v2)*v4,cos(v2)*v4,sin(v1)*v3);
}

template<typename T, typename V> V operator*(const quaternion<T>& q, const V& x)
{
  T aa = q[0]*q[0];  T bb = q[1]*q[1];
  T cc = q[2]*q[2];  T dd = q[3]*q[3];
  T ab = q[0]*q[1];  T ac = q[0]*q[2];
  T ad = q[0]*q[3];  T bc = q[1]*q[2];
  T bd = q[1]*q[3];  T cd = q[2]*q[3];

  V r(3);
  r[0] = (aa+bb-cc-dd)*x[0]  +2.0*(ad+bc)*x[1]  -2.0*(ac-bd)*x[2];
  r[1] =  -2.0*(ad-bc)*x[0]+(aa-bb+cc-dd)*x[1]  +2.0*(ab+cd)*x[2];
  r[2] =   2.0*(ac+bd)*x[0]  -2.0*(ab-cd)*x[1]+(aa-bb-cc+dd)*x[2];
  return r;
}

#endif
```

This file declares constant quaternion objects that represent the members of the 11 groups of proper crystal symmetry operators. Each set of quaternions is defined in a namespace that corresponds to its Schoenflies symbol.

```
// symmetry.hpp
// Proper symmetry operators for all crystallographic point groups
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef SYMMETRY
#define SYMMETRY
#include"quaternion.hpp"

namespace symmetry{

  namespace O{
    const double val = 0.70710678118654752440;
    const int operators = 24;
    const quaternion<double> O[24] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0,  0.0,  1.0,  0.0),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>( 0.5,  0.5,  0.5,  0.5),
      quaternion<double>( 0.5, -0.5,  0.5,  0.5),
      quaternion<double>( 0.5,  0.5, -0.5,  0.5),
      quaternion<double>( 0.5,  0.5,  0.5, -0.5),
      quaternion<double>( 0.5, -0.5, -0.5,  0.5),
      quaternion<double>( 0.5, -0.5,  0.5, -0.5),
      quaternion<double>( 0.5,  0.5, -0.5, -0.5),
      quaternion<double>( 0.5, -0.5, -0.5, -0.5),
      quaternion<double>( val,  val,  0.0,  0.0),
      quaternion<double>( val, -val,  0.0,  0.0),
      quaternion<double>( val,  0.0,  val,  0.0),
      quaternion<double>( val,  0.0, -val,  0.0),
      quaternion<double>( val,  0.0,  0.0,  val),
      quaternion<double>( val,  0.0,  0.0, -val),
      quaternion<double>( 0.0,  val,  val,  0.0),
      quaternion<double>( 0.0,  val,  0.0,  val),
      quaternion<double>( 0.0,  0.0,  val,  val),
      quaternion<double>( 0.0, -val,  val,  0.0),
      quaternion<double>( 0.0,  val,  0.0, -val),
      quaternion<double>( 0.0,  0.0,  val, -val)
    };
  }

  namespace T{
    const int operators = 12;
    const quaternion<double> O[12] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0,  0.0,  1.0,  0.0),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>( 0.5,  0.5,  0.5,  0.5),
      quaternion<double>( 0.5, -0.5,  0.5,  0.5),
      quaternion<double>( 0.5,  0.5, -0.5,  0.5),
      quaternion<double>( 0.5,  0.5,  0.5, -0.5),
      quaternion<double>( 0.5, -0.5, -0.5,  0.5),
      quaternion<double>( 0.5, -0.5,  0.5, -0.5),
      quaternion<double>( 0.5,  0.5, -0.5, -0.5),
      quaternion<double>( 0.5, -0.5, -0.5, -0.5)
    };
  }

  namespace D6{
    const double val = 0.86602540378443864676;
    const int operators = 12;
    const quaternion<double> O[12] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( val,  0.0,  0.0,  0.5),
      quaternion<double>( 0.5,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>(-0.5,  0.0,  0.0,  val),
      quaternion<double>(-val,  0.0,  0.0,  0.5),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0,  val,  0.5,  0.0),
      quaternion<double>( 0.0,  0.5,  val,  0.0),
      quaternion<double>( 0.0,  0.0,  1.0,  0.0),
      quaternion<double>( 0.0, -0.5,  val,  0.0),
      quaternion<double>( 0.0, -val,  0.5,  0.0)
    };
  }

  namespace C6{
    const double val = 0.86602540378443864676;
    const int operators = 6;
    const quaternion<double> O[6] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( val,  0.0,  0.0,  0.5),
      quaternion<double>( 0.5,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>(-0.5,  0.0,  0.0,  val),
      quaternion<double>(-val,  0.0,  0.0,  0.5)
    };

  }

  namespace D4{
    const double val = 0.70710678118654752440;
    const int operators = 8;
    const quaternion<double> O[8] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( val,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>(-val,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0,  val,  val,  0.0),
      quaternion<double>( 0.0,  0.0,  1.0,  0.0),
      quaternion<double>( 0.0, -val,  val,  0.0)
    };
  }

  namespace C4{
    const double val = 0.70710678118654752440;
    const int operators = 4;
    const quaternion<double> O[4] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( val,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0),
      quaternion<double>(-val,  0.0,  0.0,  val)
    };
  }

  namespace D3{
    const double val = 0.86602540378443864676;
    const int operators = 6;
    const quaternion<double> O[6] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.5,  0.0,  0.0,  val),
      quaternion<double>(-0.5,  0.0,  0.0,  val),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0, -0.5,  val,  0.0),
      quaternion<double>( 0.0, -0.5, -val,  0.0)
    };
  }

  namespace C3{
    const double val = 0.86602540378443864676;
    const int operators = 3;
    const quaternion<double> O[3] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.5,  0.0,  0.0,  val),
      quaternion<double>(-0.5,  0.0,  0.0,  val)
    };
  }

  namespace D2{
    const int operators = 4;
    const quaternion<double> O[4] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0),
      quaternion<double>( 0.0,  0.0,  1.0,  0.0),
      quaternion<double>( 0.0,  0.0,  0.0,  1.0)
    };
  }

  namespace C2{
    const int operators = 2;
    const quaternion<double> O[2] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0),
      quaternion<double>( 0.0,  1.0,  0.0,  0.0)
    };
  }

  namespace C1{
    const int operators = 1;
    const quaternion<double> O[1] = {
      quaternion<double>( 1.0,  0.0,  0.0,  0.0)
    };
  }
}

#endif
```

This code contains functions for computing the disorientation angle of a quaternion rotation, as well as a function that computes the random distribution of disorientation angles based on crystal symmetry.

```
// disorientation.hpp
// Functions to calculate disorientation, etc.
// Note: an expression such as q1*q2 implies that rotation q2 follows rotation q1
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef DISORIENTATION
#define DISORIENTATION
#include"symmetry.hpp"
#include<string>
#include<cmath>

template<typename T> T disorientation(const quaternion<T>& q, std::string symmetry = "O")
{
  if (symmetry=="C1") {
    using namespace symmetry::C1;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="C2") {
    using namespace symmetry::C2;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="C3") {
    using namespace symmetry::C3;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="C4") {
    using namespace symmetry::C4;
    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="C6") {
    using namespace symmetry::C6;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="D2") {
    using namespace symmetry::D2;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="D3") {
    using namespace symmetry::D3;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="D4") {
    using namespace symmetry::D4;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="D6") {
    using namespace symmetry::D6;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="T") {
    using namespace symmetry::T;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  else if (symmetry=="O") {
    using namespace symmetry::O;

    T disorientation = 2.0*M_PI;
    for (int i=0; i<operators; i++) {
      quaternion<T> a = quaternion<T>(O[i])*q;
      for (int j=0; j<operators; j++) {
        quaternion<T> b = a*quaternion<T>(O[j]);
        T angle1 = 2.0*acos(b[0]);
        T angle2 = 2.0*M_PI-angle1;
        if (angle1<disorientation) disorientation = angle1;
        if (angle2<disorientation) disorientation = angle2;
      }
    }
    return disorientation;
  }

  return 0.0;
}

template<typename T>
T disorientation(const quaternion<T>& q1, const quaternion<T>& q2, std::string symmetry = "O")
{
  return disorientation(conj(q1)*q2,symmetry);
}

template<typename T> T alpha(T r, T R = 1.0)
```

```
  {return acos(R/r);}

template<typename T> T C(T a, T b, T c)
  {return acos((cos(c)-cos(a)*cos(b))/(sin(a)*sin(b)));}

template<typename T> T S1(T a)
  {return 2.0*M_PI*(1.0-cos(a));}

template<typename T> T S2(T a, T b, T c)
  {return 2.0*(M_PI-C(a,b,c)-cos(a)*C(c,a,b)-cos(b)*C(b,c,a));}

template<typename T> T mackenzie(T angle, std::string symmetry = "O")
{
  if (angle<0.0 || angle>M_PI) return 0.0;

  if (symmetry=="C1") {
    static const T m  = 1.0;
    static const T K  = m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = tan(0.5*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="C2") {
    static const T m  = 2.0;
    static const T K  = m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = tan(0.5*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="C3") {
    static const T m  = 3.0;
    static const T K  = m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = tan(0.5*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="C4") {
    static const T m  = 4.0;
    static const T K  = m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = tan(0.5*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="C6") {
    static const T m  = 6.0;
    static const T K  = m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = tan(0.5*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi  = 0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="D2") {
    static const T m  = 2.0;
    static const T K  = 2.0*m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = 1.0;
    static const T r3 = sqrt(1.0+r1*r1);
    static const T r4 = sqrt(1.0+2.0*r1*r1);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi += -2.0*m*S1(alpha(r));
    if (r>r3) chi +=  4.0*m*S2(alpha(r,r1),alpha(r),0.5*M_PI)
                     +2.0*m*S2(alpha(r),alpha(r),M_PI/m);
    if (r>r4) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="D3") {
    static const T m  = 3.0;
    static const T K  = 2.0*m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = 1.0;
    static const T r3 = sqrt(1.0+r1*r1);
    static const T r4 = sqrt(1.0+2.0*r1*r1);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi += -2.0*m*S1(alpha(r));
    if (r>r3) chi +=  4.0*m*S2(alpha(r,r1),alpha(r),0.5*M_PI)
                     +2.0*m*S2(alpha(r),alpha(r),M_PI/m);
    if (r>r4) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="D4") {
    static const T m  = 4.0;
    static const T K  = 2.0*m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = 1.0;
    static const T r3 = sqrt(1.0+r1*r1);
    static const T r4 = sqrt(1.0+2.0*r1*r1);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi += -2.0*m*S1(alpha(r));
    if (r>r3) chi +=  4.0*m*S2(alpha(r,r1),alpha(r),0.5*M_PI)
                     +2.0*m*S2(alpha(r),alpha(r),M_PI/m);
    if (r>r4) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="D6") {
    static const T m  = 6.0;
    static const T K  = 2.0*m/180.0*M_PI;
    static const T r1 = tan(M_PI/(2.0*m));
    static const T r2 = 1.0;
    static const T r3 = sqrt(1.0+r1*r1);
    static const T r4 = sqrt(1.0+2.0*r1*r1);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -2.0*S1(alpha(r,r1));
    if (r>r2) chi += -2.0*m*S1(alpha(r));
    if (r>r3) chi +=  4.0*m*S2(alpha(r,r1),alpha(r),0.5*M_PI)
                     +2.0*m*S2(alpha(r),alpha(r),M_PI/m);
    if (r>r4) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="T") {
    static const T K  = 12.0/180.0*M_PI;
    static const T r1 = sqrt(3.0)/3.0;
    static const T r2 = sqrt(2.0)/2.0;
    static const T r3 = tan(0.25*M_PI);

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -8.0*S1(alpha(r,r1));
    if (r>r2) chi += 12.0*S2(alpha(r,r1),alpha(r,r1),acos(1.0/3.0));
    if (r>r3) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  else if (symmetry=="O") {
    static const T K   = 24.0/180.0*M_PI;
    static const T r1  = sqrt(2.0)-1.0;
    static const T r2  = sqrt(3.0)/3.0;
    static const T r3  = 2.0-sqrt(2.0);
    static const T r4  = sqrt(23.0-16.0*sqrt(2.0));

    T r = tan(0.5*angle);
    T chi = 4.0*M_PI;
    if (r>r1) chi += -6.0*S1(alpha(r,r1));
    if (r>r2) chi += -8.0*S1(alpha(r,r2));
    if (r>r3) chi += 12.0*S2(alpha(r,r1),alpha(r,r1),0.5*M_PI)
                    +24.0*S2(alpha(r,r1),alpha(r,r2),acos(r2));
    if (r>r4) chi  =  0.0;
    T value = K/(2.0*M_PI*M_PI)*sin(0.5*angle)*sin(0.5*angle)*chi;
    return fabs(value);
  }

  return 0.0;
}

#endif
```

A.3 Simulation

This file contains user-defined energy and mobility functions, as well as orientation and property tables, and can be used with an simulation method.

```
// anisotropy.hpp
// Orientations, misorientations, energy and mobility functions
// Questions/comments to jgruber@andrew.cmu.edu

#ifndef ANISOTROPY
#define ANISOTROPY
#include<fstream>
#include"disorientation.hpp"
#include"rotations.hpp"

namespace anisotropy{
  std::map<int,std::map<int,float> > angle;
  std::vector<quaternion<float> > g;
}

void read_texture(const char* filename)
{
  std::ifstream input(filename);
  using namespace anisotropy;

  // read texture from ascii text file
  while (1) {
    float  phi1, PHI, phi2;
    input>>phi1>>PHI>>phi2;
    if (input.eof()) break;
    g.push_back(euler(phi1,PHI,phi2));
  }
}


float E(int i, int j, std::string symmetry = "O")
{
  // trivial case: no boundary
  if (i==j) return 0.0;

  // uncomment for isotropy
  //return 1.0;

  // compute disorientation angle, if necessary
  using namespace anisotropy;
  int a = (i<j ? i : j);
  int b = (i<j ? j : i);
  if (angle[a][b]==0.0)
    angle[a][b] = 180.0/M_PI*disorientation(g[a],g[b]);

  // compute energy
  float energy = 1.0;

  // Read-Shockley functional form
  if (angle[a][b]<15.0) energy = angle[a][b]/15.0*(1.0-log(angle[a][b]/15.0));
  //if (angle[a][b]<30.0) energy = angle[a][b]/30.0*(1.0-log(angle[a][b]/30.0));
  //if (angle[a][b]<45.0) energy = angle[a][b]/45.0*(1.0-log(angle[a][b]/45.0));

  // step function
  //if (angle[a][b]<30.0) energy = 0.6;

  return energy;
}

float M(int i, int j)
{
  // trivial case: no boundary
  if (i==j) return 0.0;
  // uncomment for isotropy
  //return 1.0;

  // compute disorientation angle, if necessary
  using namespace anisotropy;
  int a = (i<j ? i : j);
  int b = (i<j ? j : i);
  if (angle[a][b]==0.0)
    angle[a][b] = 180.0/M_PI*disorientation(g[a],g[b]);

  // compute mobililty
  float mobility = 1.0;

  // Read-Shockley functional form
  if (angle[a][b]<15.0) mobility = angle[a][b]/15.0*(1.0-log(angle[a][b]/15.0));
  //if (angle[a][b]<30.0) mobility = angle[a][b]/30.0*(1.0-log(angle[a][b]/30.0));
  //if (angle[a][b]<45.0) mobility = angle[a][b]/45.0*(1.0-log(angle[a][b]/45.0));

  // step function
  //if (angle[a][b]<30.0) mobility = 0.6;

  return mobility;
}

#endif
```

This file contains the **MCgrid2D** and **MCgrid3D** update functions. These functions define the physical behavior of the Monte Carlo simulation, which in this case is anistropic curvature driven interface motion.

```
// MCgrid.anisotropy.hpp
// Generic algorithms for Monte Carlo methods
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef MCGRID_UPDATE
#define MCGRID_UPDATE
#include"MCgrid.hpp"
#include"anisotropy.hpp"

namespace MC{

void MCgrid2D::update(int steps)
{
  MCgrid2D& grid = *this;
  int n = nx[0]*nx[1];
  const float kT = 0.7;
  using namespace anisotropy;

  for (int step=0; step<steps; step++)
    for (int h=0; h<n; h++) {
      int x = rand()%nx[0];
      int y = rand()%nx[1];
      int spin1 = grid[x][y];

      std::vector<int> neighbors = nonzero(x,y);
      int spin2 = neighbors[rand()%neighbors.size()];

      if (spin1!=spin2) {
        float dE = -E(spin1,spin2);
        for (int i=-1; i<=1; i++)
          for (int j=-1; j<=1; j++) {
            int spin = grid.neighbor(x,y,i,j);
            dE += E(spin,spin2)-E(spin,spin1);
          }
        float num = static_cast<float>(rand())/static_cast<float>(RAND_MAX);
        if (dE<=0.0 && num<M(spin1,spin2)*E(spin1,spin2)) grid[x][y] = spin2;
        if (dE>0.0 && num<M(spin1,spin2)*E(spin1,spin2)*exp(-dE/(kT*E(spin1,spin2)))) grid[x][y] = spin2;
      }
    }
}

void MCgrid3D::update(int steps)
{
  MCgrid3D& grid = *this;
  int n = nx[0]*nx[1]*nx[2];
  const float kT = 1.5;
  using namespace anisotropy;

  for (int step=0; step<steps; step++)
    for (int h=0; h<n; h++) {
      int x = rand()%nx[0];
      int y = rand()%nx[1];
      int z = rand()%nx[2];
      int spin1 = grid[x][y][z];

      std::vector<int> neighbors = nonzero(x,y,z);
      int spin2 = neighbors[rand()%neighbors.size()];

      if (spin1!=spin2) {
        float dE = -E(spin1,spin2);
        for (int i=-1; i<=1; i++)
          for (int j=-1; j<=1; j++)
            for (int k=-1; k<=1; k++) {
              int spin = grid.neighbor(x,y,z,i,j,k);
              dE += E(spin,spin2)-E(spin,spin1);
            }
        float num = static_cast<float>(rand())/static_cast<float>(RAND_MAX);
        if (dE<=0.0 && num<M(spin1,spin2)*E(spin1,spin2)) grid[x][y][z] = spin2;
        if (dE>0.0 && num<M(spin1,spin2)*E(spin1,spin2)*exp(-dE/(kT*E(spin1,spin2)))) grid[x][y][z] = spin2;
      }
    }
}

} // namespace MC

#endif
```

This file contains the **sparsePF2D** and **sparsePF3D** update functions. These functions define the physical behavior of the phase field simulation, which in this case is anistropic curvature driven interface motion.

```
// sparsePF.anisotropy.hpp
// Generic algorithms for phase field methods
// Questions/comments to jgruber@andrew.cmu.edu (Jason Gruber)

#ifndef SPARSEPF_UPDATE
#define SPARSEPF_UPDATE
#include"sparsePF.hpp"
#include"anisotropy.hpp"

namespace MMSP{

void sparsePF2D::update(int steps, int id, int np, int ng)
{
  sparsePF2D& grid = *this;
  using namespace anisotropy;

  const float w  = 6.0;
  const float a  = 4.0*w/M_PI/M_PI;
  const float b  = 4.0/w;
  const float c  = 1.0/a;
  const float dt = 0.02;

  for (int step=0; step<steps; step++) {
    sparsePF2D update(nx[0],nx[1]);

    for (int x=ng; x<nx[0]-ng; x++)
      for (int y=0; y<nx[1]; y++) {
        std::vector<int> p = nonzero(x,y);
        int N = p.size();

        if (N==1)
          update[x][y][p[0]] = 1.0;

        else {
          std::vector<float> F(N,0.0);
          for (int i=0; i<N; i++)
            for (int j=i+1; j<N; j++) {
              F[i] += E(p[i],p[j])*(a*laplacian(x,y,p[j])+b*grid[x][y][p[j]]);
              F[j] += E(p[i],p[j])*(a*laplacian(x,y,p[i])+b*grid[x][y][p[i]]);
            }

          std::vector<float> H(N,0.0);
          for (int i=0; i<N; i++)
            for (int j=i+1; j<N; j++) {
              H[i] += c*M(p[i],p[j])*(F[i]-F[j]);
              H[j] += c*M(p[i],p[j])*(F[j]-F[i]);
            }

          float sum = 0.0;
          for (int i=0; i<N; i++) {
            const float epsilon = 1.0e-6;
            float value = grid[x][y][p[i]]-dt*H[i]/N;
            if (value>1.0) value = 1.0;
            if (value<0.0) value = 0.0;
            if (value>epsilon) update[x][y][p[i]] = value;
            sum += value;
          }

          float rsum = 1.0/sum;
          for (int i=0; i<N; i++)
            if (update[x][y][p[i]]>0.0)
              update[x][y][p[i]] *= rsum;
        }
      }
    grid.ghostswap(id,np,ng);
    grid.swap(update);
  }
}

void sparsePF3D::update(int steps, int id, int np, int ng)
{
  sparsePF3D& grid = *this;
  using namespace anisotropy;

  const float w  = 6.0;
  const float a  = 4.0*w/M_PI/M_PI;
  const float b  = 4.0/w;
  const float c  = 1.0/a;
  const float dt = 0.02;

  for (int step=0; step<steps; step++) {
    sparsePF3D update(nx[0],nx[1],nx[2]);

    for (int x=ng; x<nx[0]-ng; x++)
      for (int y=0; y<nx[1]; y++)
        for (int z=0; z<nx[2]; z++) {
          std::vector<int> p = nonzero(x,y,z);
          int N = p.size();

          if (N==1)
            update[x][y][z][p[0]] = 1.0;

          else {
            std::vector<float> F(N,0.0);
            for (int i=0; i<N; i++)
              for (int j=i+1; j<N; j++) {
                F[i] += E(p[i],p[j])*(a*laplacian(x,y,z,p[j])+b*grid[x][y][z][p[j]]);
                F[j] += E(p[i],p[j])*(a*laplacian(x,y,z,p[i])+b*grid[x][y][z][p[i]]);
              }

            std::vector<float> H(N,0.0);
            for (int i=0; i<N; i++)
              for (int j=i+1; j<N; j++) {
                H[i] += c*M(p[i],p[j])*(F[i]-F[j]);
                H[j] += c*M(p[i],p[j])*(F[j]-F[i]);
              }

            float sum = 0.0;
            for (int i=0; i<N; i++) {
              const float epsilon = 1.0e-6;
              float value = grid[x][y][z][p[i]]-dt*H[i]/N;
              if (value>1.0) value = 1.0;
              if (value<0.0) value = 0.0;
              if (value>epsilon) update[x][y][z][p[i]] = value;
              sum += value;
            }

            float rsum = 1.0/sum;
            for (int i=0; i<N; i++)
              if (update[x][y][z][p[i]]>0.0)
                update[x][y][z][p[i]] *= rsum;
          }
        }
      }
    grid.ghostswap(id,np,ng);
    grid.swap(update);
  }
}

} // namespace MMSP

#endif
```

A.4 Analysis

This file contains a single program that extracts relevant grain and interface data from Monte Carlo grids.

```
// analysis.cpp
// MCgrid grain and interface analysis
// Questions/comments to jgruber@andrew.cmu.edu

#include"MCgrid.hpp"
using namespace MMSP;

#include"anisotropy.hpp"
using namespace anisotropy;

int main(int argc, char* argv[])
{
  // get grid dimension
  std::ifstream input(argv[1]);
  int dim;
  input.read(reinterpret_cast<char*>(&dim),sizeof(dim));
  input.close();

  // get texture data
  read_texture(argv[2]);

  // read symmetry (optional)
  std::string symmetry = "O";
  if (argc>3) symmetry = argv[3];

  // data structures: maps let us use
  // only the memory we need, automatically
  std::map<int,int> grains;
  std::map<int,bool> edge;
  std::map<int,std::map<int,int> > area;

  if (dim==2) {
    // perform analysis for 2D grids
    MCgrid2D grid(argv[1]);
    int nx = grid.size(0);
    int ny = grid.size(1);
    bool px = grid.boundary(0);
    bool py = grid.boundary(1);

    for (int x=0; x<nx; x++)
    for (int y=0; y<ny; y++) {
      int IDa = grid[x][y];
      grains[IDa] += 1;
      if (!px && (x==0 || x==nx-1)) edge[IDa] = true;
      if (!py && (y==0 || y==ny-1)) edge[IDa] = true;
      for (int i=0; i<=1; i++)
      for (int j=0; j<=1; j++) {
        int IDb = grid.neighbor(x,y,i,j);
        if (IDa!=IDb) {
          int a = (IDa<IDb ? IDa : IDb);
          int b = (IDa<IDb ? IDb : IDa);
          area[a][b] += 1;
          if (angle[a][b]==0.0)
            angle[a][b] = 180.0/M_PI*disorientation(g[a],g[b],symmetry);
        }
      }
    }
  }

  if (dim==3) {
    // perform analysis for 3D grids
    MCgrid3D grid(argv[1]);
    int nx = grid.size(0);
    int ny = grid.size(1);
    int nz = grid.size(2);
    bool px = grid.boundary(0);
    bool py = grid.boundary(1);
    bool pz = grid.boundary(2);

    for (int x=0; x<nx; x++)
    for (int y=0; y<ny; y++)
    for (int z=0; z<nz; z++) {
      int IDa = grid[x][y][z];
      grains[IDa] += 1;
      if (!px && (x==0 || x==nx-1)) edge[IDa] = true;
      if (!py && (y==0 || y==ny-1)) edge[IDa] = true;
      if (!pz && (z==0 || z==nz-1)) edge[IDa] = true;
      for (int i=0; i<=1; i++)
      for (int j=0; j<=1; j++)
      for (int k=0; k<=1; k++) {
        int IDb = grid.neighbor(x,y,z,i,j,k);
        if (IDa!=IDb) {
          int a = (IDa<IDb ? IDa : IDb);
          int b = (IDa<IDb ? IDb : IDa);
          area[a][b] += 1;
          if (angle[a][b]==0.0)
            angle[a][b] = 180.0/M_PI*disorientation(g[a],g[b],symmetry);
        }
      }
    }
  }

  // write grain data
  // grain ID, grain volume (area), edge flag
  std::ofstream gfile("grains");
  std::map<int,int>::iterator git;
  for (git=grains.begin(); git!=grains.end(); git++) {
    int ID = (*git).first;
    int volume = (*git).second;
    gfile<<ID<<" "<<volume<<" "<<edge[ID]<<std::endl;
  }
  gfile.close();

  // write interface data
  // grain IDa, grain IDb, angle, area (length), edge flag
  std::ofstream ifile("interfaces");
  std::map<int,std::map<int,float> >::iterator ait;
  for (ait=angle.begin(); ait!=angle.end(); ait++) {
    int a = (*ait).first;
    std::map<int,float>::iterator bit;
    for (bit=angle[a].begin(); bit!=angle[a].end(); bit++) {
      int b = (*bit).first;
      float theta = (*bit).second;
      ifile<<a<<" "<<b<<" "<<theta<<" "<<area[a][b];
      ifile<<" "<<(edge[a]&&edge[b])<<std::endl;
    }
  }
  ifile.close();
}
```