INCORPORATING ANISOTROPIC GRAIN BOUNDARY ENERGY INTO A PHASE FIELD MODEL OF URANIUM DIOXIDE

by

Jared Tolliver

A senior thesis submitted to the faculty of

Brigham Young University-Idaho

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics

Brigham Young University-Idaho

April 2016

BRIGHAM YOUNG UNIVERSITY-IDAHO

DEPARTMENT APPROVAL

of a senior thesis submitted by

Jared Tolliver

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Date	Evan Hansen, Advisor
Date	Todd Lines, Research Coordinator
Date	Stephen McNeil, Chair
Date	Richard Hatt, Committee Member
Date	Lance Nelson, Committee Member

ABSTRACT

Understanding the microstructure of nuclear fuels is important because the microstructure affects many important properties such as thermal conductivity, fission gas release and mechanical stability of fuel materials. Idaho National Laboratory's (INL's) mesoscale fuels performance code MARMOT currently assumes that the grain boundary energy and mobility are isotropic when calculating the grain boundary. They are known to depend on how the two grains are oriented with respect to each other and the grain boundary. We have incorporated the anisotropic nature of the grain boundary energy into MARMOT to increase the accuracy of the simulation. Using a five degree of freedom model to predict the grain boundary energies for , we added code to MARMOT to calculate the grain boundary energy at each boundary instead of assuming the grain boundary energy is the same everywhere. We preformed a few test cases such as a single circle grain, a small array of hexagon grains, and a many grain system. From these test cases we concluded that another term needed to be added to the phase field equations in order for the free energy to decrease. Another study is being performed to validate our model.

ACKNOWLEDGMENTS

I would like to thank the Department of Energy's Visiting Faculty Program for providing the funding for this research. I would also like to thank Mike Tonks, Yongfeng Zhang, Daniel Schwen and Evan Hansen for answering my questions I had related to this research and providing feedback and direction.

Contents

Chapter 1

Introduction

There has been interest recently in developing sustainable sources of energy other than fossil fuels. Nuclear fission is one of these sustainable sources of energy. Being able to computationally model nuclear reactors allows us to predict how reactors will behave in normal as well as in accident conditions, and for new reactor designs. To do this researchers perform multiscale modeling. Multiscale modeling is where first principles simulations inform atomistic simulations which inform mesoscale models which in turn inform engineering scale models(1). Different scales are dominated by different physics. This is used in the modeling of nuclear reactors is the modeling of grain growth and its effects. First, at the scale of electrons, the Schrödinger equation is solved to describe the interaction between adjacent atoms. Then, molecular dynamics uses knowledge about these interactions to model how atoms behave at grain boundaries. Next, molecular dynamics simulations give the parameters to be used in grain growth simulations. Finally in the engineering scale, the microstructure is produced in grain growth simulations which affects material properties such as thermal conductivity.

There are many other effects that need this kind of multiscale modeling. This is a large endeavour and my project was to improve a small portion of the mesoscale model. More specifically my project was to add anisotropic grain boundary energy in MARMOT (Idaho National Laboratory's mesoscale fuel performance code). MARMOT uses MOOSE(Multi-physics Object Oriented Simulation Environment) as a framework to solve the equations related to mesoscale modeling(2).

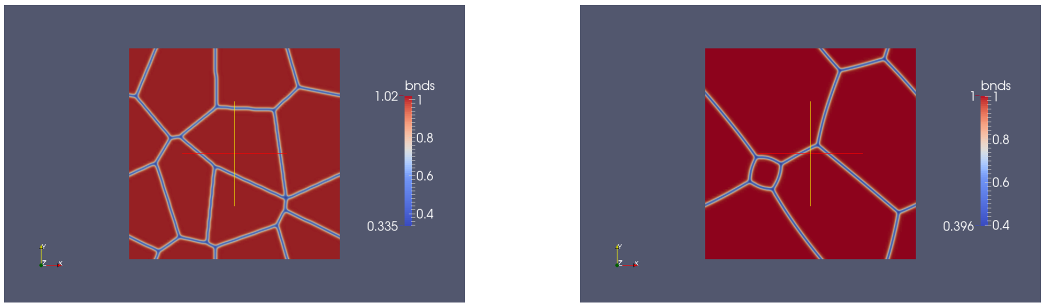
1.1 Grain Growth

Many metals and ceramics are polycrystalline. Polycrystalline materials are made up of many regions that are each a single crystal. Grains are these single crystal regions. Grains are usually between 1 μm and 100 μm in size. This length scale is mesoscale(3).

A grain boundary is where two or more grains meet. These grain boundaries add energy to the system because they are lattice imperfections. In other words, they are made up of broken atomic bonds. Factors that effect the amount of energy in the boundary are the orientation of the two grains with respect to each other and to the boundary. To reduce the energy, atoms tend to move from one grain to another.

We are interested in how grains evolve in materials because the size, shape and orientation of the grains affects material properties such as thermal conduction, mechanical properties(3). Our driving motivation in this research is to more accurately model nuclear reactors. Our part in this is to model how the grains evolve (grain growth) to more effectively in order to calculate these material properties to use in larger scale simulations.

The current model is MARMOT which assumes that the grain boundary energy is isotropic, meaning the energy is the same for every grain. This is called the isotropic case. The goal of my project was to calculate the grain boundary energy using the orientation of the grains. We use this to more accurately calculate the evolution of the grains. This is the anisotropic case. Figure 1.1 is an example of what a small 8 grain simulation might look like before and after grain growth.



(a) before grain growth **(b)** after grain growth

Figure 1.1 A small 8 grain simulation in MARMOT.

1.1.1 Phase Field

Because different scales use different physics, they require different models. To numerically model grain growth we use a phase field model. In a phase field model we label each grain with an order parameter $\eta_i(\vec{r}, t)$. The order parameter is used to keep track of what grains are where. An order parameter is equal to 1 inside the grain it represents and 0 outside of the grain. In the grain boundary it transitions smoothly from 1 to 0, as seen in Figure 1.2.

The development of these order parameter is given by the Allen-Cahn equation. We apply the Allen-Cahn equations to each order parameter to get a system of partial differential equations. The Allen-Cahn equation is $\frac{\partial \eta_i(\vec{r},t)}{\partial t} = -L\frac{\delta F}{\delta \eta_i}$, where F is the total free energy and $\frac{\delta F}{\delta \eta_i}$ is the functional derivative of the free energy with respect to one of the order parameters(4). The system evolves according to the equations until the free energy is minimized.

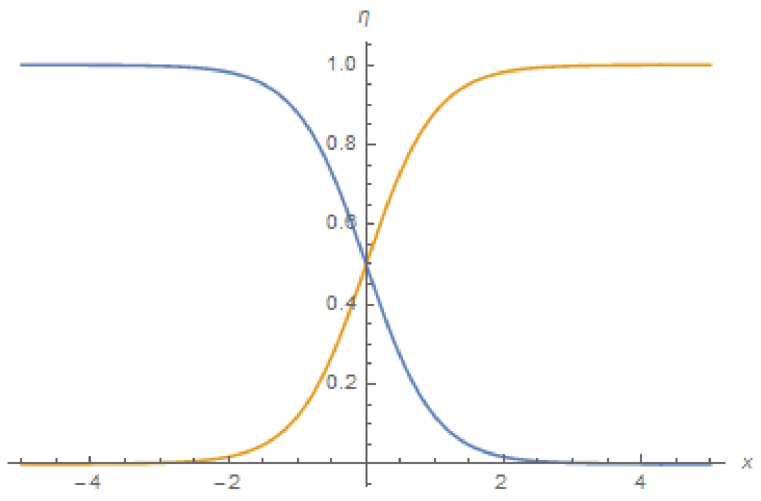


Figure 1.2 The order parameters at a grain boundary.

The free energy is given by

$$F = \int_R f(\eta_1, \eta_2, \eta_3, ..., \eta_p, \nabla\eta_1, \nabla\eta_2, \nabla\eta_3, ..., \nabla\eta_p)dV,$$

where f is the free energy density and is given by

$$f = \mu\Big(\sum_{i=1}^{p}\left(\frac{\eta_i^4}{4} - \frac{\eta_i^2}{2}\right) + \gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2 + \frac{1}{4}\Big) + \frac{\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2.$$

The term multiplied by μ is the bulk energy. The bulk free energy as a function of two order parameters is seen in Figure 1.3. The minimum is when one order parameter is one and the other is zero. There is a saddle point when both order parameters are 1/2. These properties ensure that the order parameters will move towards one being one and the other order parameter being zero. The last term is the gradient energy term which represents the energy due to the profile of a grain boundary. This term ensures that grain boundaries have the proper width(instead of no width). p is the number of grains, μ is the bulk energy coefficient, γ is another constant that affects the bulk grain boundary energy and is found to match correct behavior when $\gamma = 1.5$, and κ is the gradient free energy coefficient.

When we take the functional derivative for the isotropic case we obtain the Allen-

Cahn equation,

$$\frac{\partial \eta_i}{\partial t} = -L\Big[\mu\Big(\eta_i^3 - \eta_i + 2\gamma\sum_{j=1}^{p}\eta_i\eta_j^2\Big) - \nabla\cdot(\kappa\nabla\eta_i)\Big].$$

Differences that occur because of the anisotropy will be discussed later in Chapter 3.

The parameters for the Allen-Cahn equation are given in Moeleans *et. al* (4) as

$$L \approx \frac{4M}{3l_{gb}}$$

$$\kappa \approx \frac{3}{4}\sigma_{gb}l_{gb}$$

$$\mu \approx \frac{3\sigma gb}{4f_{0,saddle}l_{gb}}$$

$$\gamma \approx 1.5$$

$$f_{0,saddle} \approx 1/8$$

Here σ_{gb} is the grain boundary energy, l_{gb} is the grain boundary width and M is a temperature dependent mobility of the grain boundaries. These equations are solved numerically in MARMOT.

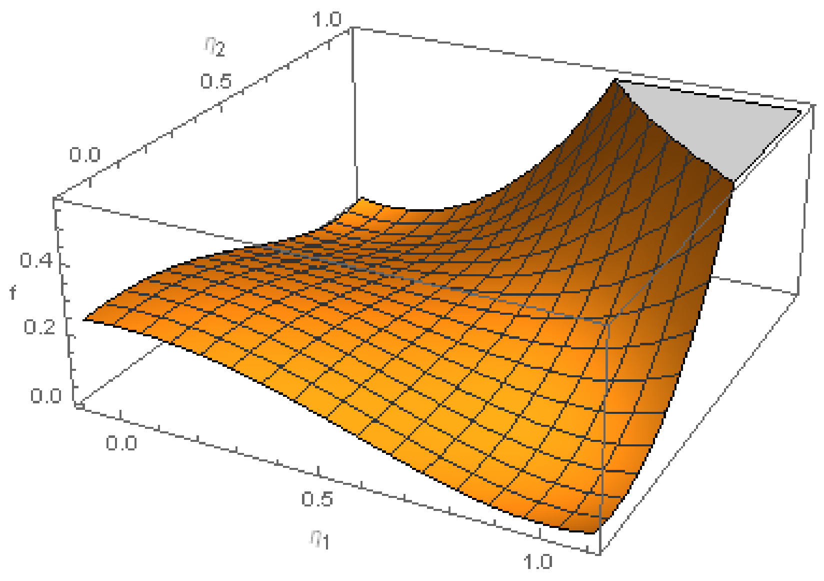


Figure 1.3 The bulk free energy as a function of two order parameters

1.1.2 Five Degree of Freedom Grain Boundary Energy Model

Grain boundaries have 5 degrees of freedom. There are two common ways to describe these 5 degrees of freedom: the misorientation/inclination description and the twist/tilt description. In the misorientation/inclination description we hold one of the grains fixed and allow the other grain to rotate with 3 degrees of freedom (this is called misorientation). The other two degrees of freedom come from rotating the grain boundary (this is called inclination)(4)(5). In the twist/tilt description the grain boundary is held in the (001) direction – or in other words the normal along the z axis – and both grains are allowed to rotate. A twist is when the rotation axis is orthogonal to the grain boundary plane. A tilt is when the rotation axis is in the plane. Anything else is a combination of twists and tilts(6) (3). A schematic of a tilt and a twist boundary is in Figure 1.4.

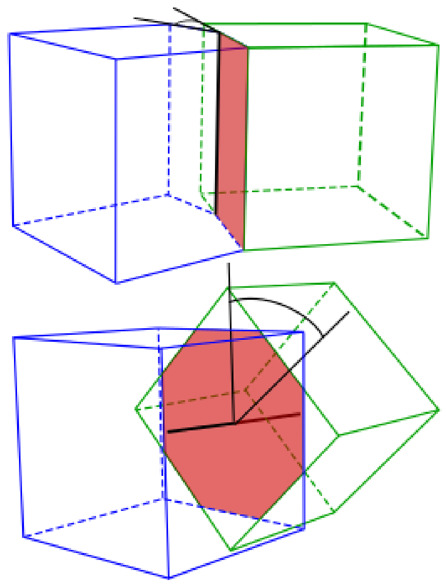


Figure 1.4 The top is a representation of a tilt grain boundary; the bottom is a representation of a twist grain boundary. Courtesy of Wikipedia, under the creative commons licence.

The grain boundary energy, σ , is a function of these 5 degrees of freedom. In the isotropic case σ is treated as constant. To improve the accuracy of the simulation we calculate the grain boundary energy at each element. To calculate the grain boundary energy we use a function developed by Vasily Bulatov(6) which used molecular dynamics to calculate the grain boundary energy of 388 grain boundaries to find a 43 parameter fit to the grain boundary energy function for 4 FCC metals (Cu, Ni, Au, and Al). Bulatov created 1D, 2D and 3D subsets and combine them to get the 5D function(a function of the 5 degrees of freedom). The 1D subsets are symmetric tilt grain boundaries (STGBs) and twist boundaries for <100>, <110> and <111> axes. The 2D subsets are asymmetric tilt grain boundaries (ATGBs) in the same set of axes. The 3D subsets are a combination of twist and tilt in one of the sets of axes.

The 1D subsets could be described with one angle ξ which for twist boundaries is the angle of rotation and for tilts it is the angle between the two grains. For ATGBs we introduce another angle η which describes how asymmetric the angle is. θ_1 is the angle between the top grain and the grain boundary and like wise with θ_2 and the bottom grain. $\xi = \theta_1 - \theta_2$ and $\eta = \theta_1 + \theta_2$ (remember since θ_2 is below the boundary it is negative). In 3D subsets we need another angle ϕ which is the angle between the rotation axis and the unit normal. ϕ is 0 for twist boundaries and $\pi/2$ for tilt boundaries. In Figure 1.5 is what the 1-D and 2-D subsets look like for 110 rotations.

All of the 1D subsets are defined as piecewise functions where each piece is a Read Shockley Wolfe (RSW) function. RSW functions take the form $\sin(\theta)(1 - a * \ln(\sin(\theta)))$. The <100> ATGB uses a power law and the energies of <100> STGB at ξ and $\pi/2 - \xi$. The <110> ATGB uses an RSW function and the energies of <100> STGB at ξ and $\pi - \xi$. There is no <111> STGB but there is an <111> ATGB which uses an RSW function for both ξ and η. All of the 3D subsets use a power law: $\epsilon_{hkl}(\xi, \eta, \phi) = \epsilon_{hkl}^{twist}(\xi)(1 - \frac{2\phi}{\pi})^{p_{hkl}^1} + \epsilon_{hkl}^{tilt}(\xi, \eta)(\frac{2\phi}{\pi})^{p_{hkl}^2}$, where p_{hkl}^1 and p_{hkl}^2 are two

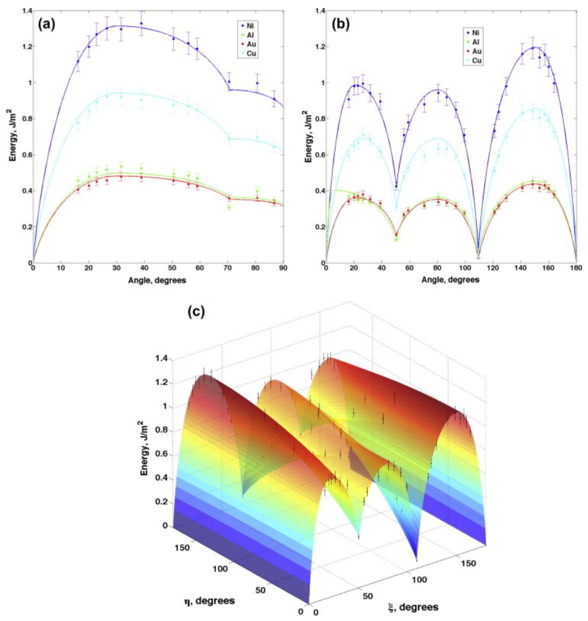


Figure 1.5 The energy of 110 rotations subsets for a) twist boundaries b) symmetric tilt boundaries and c) asymmetric tilt boundaries. This figure is taken from Bulatov's paper(6).

of the 43 parameters.

Bulatov then used weighing functions to interpolate between the different idealized rotations. To do interpolation, distance has to be defined. d_3, the difference in the misorientions, is define as $d_3 = 2\sin(\delta/2)$ where δ is the residual angle after doing the actual rotation and then doing the inverse idealized rotation. d_2 is the inclination distance defined as $d_2 = 2 - m_1 \cdot n_1 - m_2 \cdot n_2$ where n_1 and n_2 are the unit normal in grain 1 reference and grain 2 reference and m_1 and m_2 are the unit normal for the idealized versions of grain 1 and 2. The weighting function is

$$w_{hkl}(d_3) = \frac{w^0_{hkl}}{\sin(\frac{\pi d_3}{2d^{max}_{hkl}})(1 - \frac{1}{2}\ln\sin(\frac{\pi d_3}{2d^{max}_{hkl}}) - 1}$$

where w_{hkl} and d^{max}_{hkl} are the weights and max distances in one of <100>, <110> or <111>. The energy would then be:

$$\epsilon = \frac{1 + \sum w_{hkl}(d_3)\epsilon_{hkl}(\xi, \eta, \phi)}{1 + \sum w_{hkl}(d_3)}\epsilon_{RGB}$$

where ϵ_{RGB} is the the energy scaling factor that the energies are scaled from. It is one of the 43 parameters.

Tim Harbison extended this model to include, a fluorite ceramic, by finding the 43 parameters for UO_2 which allows us to use the anisotropic model for grain growth in UO_2. (5)

1.2 MOOSE

MOOSE is a framework for solving coupled non-linear partial differential equations. It was developed at Idaho National Laboratory (INL) in 2009. It uses the finite element method which produces a piece-wise continuous solution to the differential equations being solved. MOOSE performs these calculations on each quadrature point.(7)

In MOOSE the calculations are partitioned into separate classes. MOOSE is very structured and has several parent classes that have different functionality. For a specific application, one would inherit from one of these classes to make their own unique class or use one of the classes that is already in MOOSE's physics modules. An input file tells MOOSE which classes to use, how to use them, and give some parameters to the classes. I will give a brief overview of the function of a few of these parent classes and how they fit into MOOSE. This is necessary in order to understand how the code that I wrote works and fits into the simulation as a whole. For more information about these parent classes visit mooseframework.org/wiki/MooseSystems.

1.2.1 Input File

In the input file we specify the variables that are used in the partial differential equations we are trying to solve. In the case of grain growth the variables are the order parameters. Each kernel describes one term in the differential equation. The kernels use the weak (integral) form of the differential equation. Obtaining the weak form of the Allen-Cahn equation is discussed in Appendix A. Below is an example of an input file.

```
[Mesh]
  # Mesh block.  Meshes can be read in or automatically generated
  type = GeneratedMesh
  dim = 2 # Problem dimension
  nx = 12 # Number of elements in the x-direction
  ny = 12 # Number of elements in the y-direction
  xmax = 1000 # maximum x-coordinate of the mesh
  ymin = 0 # minimum y-coordinate of the mesh
  ymax = 1000 # maximum y-coordinate of the mesh
  elem_type = QUAD4 # Type of elements used in the mesh
  uniform_refine = 3 # Initial uniform refinement of the mesh
[]

[GlobalParams]
  # Parameters used by several kernels that are defined globally to simplify input file
  op_num = 14 # Number of order parameters used
  var_name_base = gr # Base name of grains
  grain_num = 36 #Number of grains
[]

[Variables]
  # Variable block, where all variables in the simulation are declared
  [./PolycrystalVariables]
  [../]
[]

[ICs]
  [./PolycrystalICs]
    [./PolycrystalVoronoiIC]
    [../]
  [../]
[]

[AuxVariables]
  # Dependent variables
  [./bnds]
    # Variable used to visualize the grain boundaries in the simulation
    order = FIRST
    family = LAGRANGE
  [../]
  [./unique_grains]
    order = CONSTANT
    family = MONOMIAL
  [../]
  [./var_indices]
    order = CONSTANT
    family = MONOMIAL
  [../]
[]

[Kernels]
  [./Polycrystal5DoFKernel]
    AnisoGBEnergy_UserObject = AnisoEnergy
  [../]
```

```
[]

[AuxKernels]
  # AuxKernel block, defining the equations used to calculate the auxvars
  [./bnds_aux]
    type = BndsCalcAux
    variable = bnds
    execute_on = timestep_end
  [../]
  [./unique_grains]
    type = FeatureFloodCountAux
    variable = unique_grains
    execute_on = timestep_end
    bubble_object = grain_tracker
    field_display = UNIQUE_REGION
  [../]
  [./var_indices]
    type = FeatureFloodCountAux
    variable = var_indices
    execute_on = timestep_end
    bubble_object = grain_tracker
    field_display = VARIABLE_COLORING
  [../]
[]

[BCs]
  # Boundary Condition block
  [./Periodic]
    [./top_bottom]
      auto_direction = 'x y' # Makes problem periodic in the x and y directions
    [../]
  [../]
[]

[Materials]
  [./CuGrGr]
    # Material properties
    type = GBAnisoEnergy
    block = 0 # Block ID (only one block in this problem)
    GBmob0 = 2.5e-6 #Mobility prefactor for Cu from Schonfelder1997
    Q = 0.23 #Activation energy for grain growth from Schonfelder 1997
    T = 450 # K   #Constant temperature of the simulation (for mobility calculation)
    wGB = 14 # nm  #Width of the diffuse GB
    AnisoGBEnergy_UserObject = AnisoEnergy
    outputs = exodus
  [../]
[]

[Postprocessors]
  # Scalar postprocessors
  [./dt]
    # Outputs the current time step
    type = TimestepSize
  [../]
[]

[UserObjects]
  [./grain_tracker]
    type = GrainTracker
    threshold = 0.2
    convex_hull_buffer = 5.0
    use_single_map = false
    enable_var_coloring = true
    condense_map_info = true
    connecting_threshold = 0.08
    flood_entity_type = elemental
    compute_op_maps = true
    execute_on = 'initial timestep_begin'
    tracking_step = 0
  [../]
  [./euler_angle_file]
    type = EulerAngleFileReader
    file_name = grn_100_rand.tex
  [../]
  [./AnisoEnergy]
    type = AnisoGBEnergyUserObject
    euler_angle_provider = euler_angle_file
    GrainTracker_object = grain_tracker
    execute_on = 'nonlinear linear'
    GBenergy_iso = 0.708
  [../]
[]

[Executioner]
  type = Transient # Type of executioner, here it is transient with an
  # adaptive time step
  scheme = bdf2 # Type of time integration (2nd order backward euler),
  #defaults to 1st order backward euler

  #Preconditioned JFNK (default)
  solve_type = 'PJFNK'

  # Uses newton iteration to solve the problem.
  petsc_options_iname = '-pc_type -pc_hypre_type -ksp_gmres_restart -mat_mffd_type'
  petsc_options_value = 'hypre boomeramg 31 ds'
  l_max_its = 25 # Max number of linear iterations
  l_tol = 1e-4 # Relative tolerance for linear solves
  nl_max_its = 25 # Max number of nonlinear iterations
  nl_rel_tol = 1e-9 # Relative tolerance for nonlienar solves
  start_time = 0.0
  end_time = 4000
  [./TimeStepper]
    type = IterationAdaptiveDT
    dt = 2.5 # Initial time step.  In this simulation it changes.
    optimal_iterations = 6 s
    growth_factor = 1.2
    cutback_factor = 0.75
  [../]
  [./Adaptivity]
    # Block that turns on mesh adaptivity.
    initial_adaptivity = 2 # Number of times mesh is adapted to initial condition
    refine_fraction = 0.7 # Fraction of high error that will be refined
    coarsen_fraction = 0.1 # Fraction of low error that will coarsened
    max_h_level = 4 # Max number of refinements used, starting from initial mesh
  [../]
[]

[Outputs]
  file_base = tracker_2D
  exodus = true # Exodus file will be outputted
  csv = true
[]
```

The mesh block tells MOOSE how fine spatially the solution will be and the size of the whole simulation. There is mesh adaptivity in MOOSE so the mesh will change size in some areas (in our case grain boundaries) to increase accuracy where it is needed. Parameters for the mesh adaptivity are in the executioner block.

There are some parameters that are used in multiple blocks. Those parameters could be put in the globalparams block so that they are only in one place but used by several blocks.

The variables block is where the variables are set up. In our case it is the order parameters. The variable names are gr0, gr1, gr2, ... The parameters declaring the names of the variables and the number of variables is in the globalparams block. The initial conditions block is where the initial values of the variables are set. The Kernels block is where the terms of the differential equation are specified. The BCs block is where the boundary conditions of each variable are set. For our simulations we usually have periodic boundary conditions. All of these blocks use the action system. This means instead of the user creating the sub-blocks explicitly, the user creates them with an action. This is where an action is specified that could create many sub-blocks at a time. In the case of the kernel action it creates a few kernels, one for each term in the differential equation, for each variable. In this example there are 4 kernels for 14 variables giving 56 kernels.

Auxiliary variables are extra variables that are not part of a differential equation that are used elsewhere in the program. Sometimes they are used in other calculation and sometimes they are used for visualization purposes. Auxiliary kernels are where the auxiliary variables are calculated.

The materials class is used to determine the material properties that are used in the kernels. These properties are calculated at each quadrature point. This makes it easy to switch between varying material properties and constant material properties. The materials class takes input parameters that are from other parts of the simulation or constants defined in the input file.

Postprocessor do some calculation and returns a value that describes the state of the system in some way. One example of a postprocessor is one that could tell the size of a specific grain. So with this postprocessor we could see how the grains change size with respect to time. There are also vector postprocessors that remove the restriction of only giving out one value; these are useful to analyze much more data.

User objects are used to do some calculations and have functionality that isn't part of another class. User objects are more customizable in that the writer defines the interface between the user object and other classes. One example of a user object is grain tracker. It allows us to use the same order parameter for multiple grains, and when two grains with the same order parameter get too close then one of the grains switches to a different order parameter. This allows us to get away with using only around 30 order parameters in a 3 dimensional simulation with hundreds of grains. This simplifies the computation by having a system of 30 equations instead of over 100 equations.

The executioner block is where details about the solver are defined. It also is where the initial timestep and timestep and mesh adaptivity are defined. The outputs block tells where to write files and what kind of files to write.

1.2.2 MARMOT

To create simulations, INL developed the MOOSE-BISON-MARMOT (MBM) suite. MOOSE is the framework where all the equations are solved. BISON has engineering scale models. MARMOT has material specific mesoscale models describing microstructure evolution(2) (1).

MOOSE has a phase-field module. In the phase field module there is an isotropic grain growth model. MARMOT has more phase-field models and is used for material specific models. Since the Bulatov's function is only for a few materials that is where our anisotropic model belongs.

Chapter 2

Auxilary Kernel Implementation

Bulatov provided a Matlab code that calculates the grain boundary energy. It takes the orientation of the two grains in a rotation matrix for each grain, which we will call orientation matrices. His code assumes a (001) boundary plane (6). We based our code off of the code he provided.

2.1 Procedures

We created an auxiliary variable that holds the grain boundary energy for each quadrature point. We wrote an auxiliary kernel that calculates the grain boundary energy. With this grain boundary energy we calculate the parameters of the Allen-Cahn equation in a material. We do this at each element at each time step. All of this calculation can be done in a material but we do most of it in the auxiliary kernel so that we could have control of when the calculation is performed. There are several solves for each time step, and material classes always do their calculations at each solve. With auxiliary kernels we could choose when to do the calculation. If we only do the grain boundary energy calculations once a time step we could have reasonable time steps. If we do the calculations every solve then the solution only converges as very small time steps.

Since Bulatov's function assumes a (001) normal to the grain boundary, we have to rotate the grains so the plane normal is at (001). We find the grain boundary normal by the direction of the gradient of one of the order parameters in the grain boundary, $\frac{\nabla\eta_i}{|\nabla\eta_i|}$. We need to rotate this vector to (001). Since a rotation around the boundary plane normal does not do anything (i.e there are two degrees of freedom for the boundary plane normal and three for a rotation matrix) there are multiple rotation matrices that could accomplish this. We used a function that was already in MOOSE that does this for us. The grain boundary energy is the same if both grains are rotated around the plane normal, so any valid rotation matrix will work. We could find the orientation of the grains in the new frame by $O' = RO$, where O' is the new orientation matrix, R is the rotation matrix, and O is the old orientation matrix. Once we have the orientation matrices in the right frame we could use Bulatov's function to calculate the grain boundary energy.

Bulatov's function only calculates the energy of a boundary with two grains. When we have a multi-junction, where 3 or more grains meet, we use a weighted average of all of the pairs. From Moelans(4) we have

$$\sigma = \frac{\sum_{i=1}^{p}\sum_{j>i}^{p}\sigma_{ij}\eta_i^2\eta_j^2}{\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2}.$$

This makes it so the energy varies smoothly from one boundary to another.

2.2 Results

In MOOSE we start with an initial grain arrangement. We start with certain initial conditions where we know what should happen. Then we can compare what should happen with what happened. We learned some valuable information about where our model was lacking.

The first test case is that of a circular grain. In the isotropic case the circle grain shrinks uniformly becoming a smaller and smaller circle. In the anisotropic case we expect that the side with the higher energy to shrink faster. This is what happened.

The second case is just that of a 2-d polycrystalline grain structure. We calculated the grain boundary energy in two simulations but only used the energy in the calculations of the grain growth in one case, the anisotropic case. We expected that the average grain boundary energy of the anisotropic case would decrease faster than in the isotropic because high energy boundaries should preferentially shrink. This was when multi-junctions were a single high energy so if a multi-junction disappears then the average grain boundary energy decreases. The average grain boundary energy of the isotropic case decreases faster, which is the opposite of what was expected. This shows that the high energy boundaries were shrinking slower in the anisotropic case instead of shrinking faster.

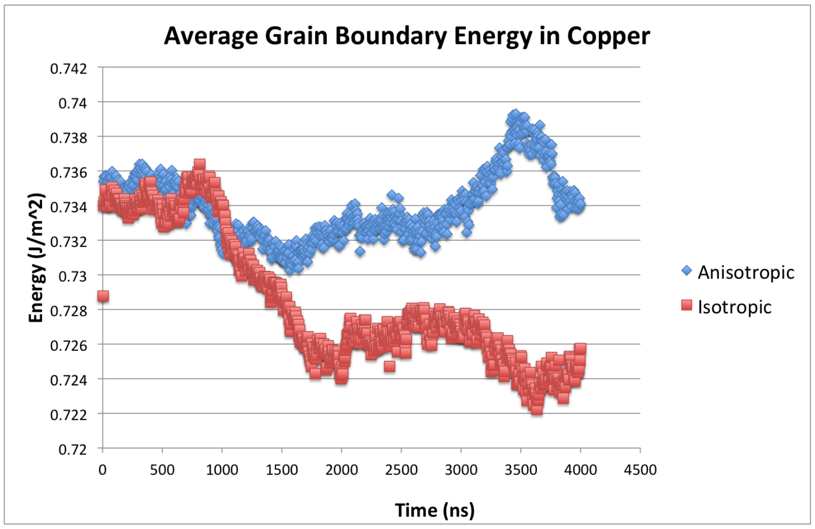


Figure 2.1 This figure shows how the average grain boundary energy goes up in the anisotropic case but not in the isotropic case. Courtesy of John-Michael Bradley. Used with permission.

To further investigate the discrepancy with the energy I created another test case to test my hypothesis that the issue was improper treatment of triple junction kinetics. Young's equation, which is an equation describing equilibrium angles of triple junctions, is $\frac{\sigma_1}{\sin\theta_{23}} = \frac{\sigma_2}{\sin\theta_{13}} = \frac{\sigma_3}{\sin\theta_{12}}$(3). So, if all of the energies are the same then all the equilibrium angles would be 120°. The third case was a grid of 4 hexagons. In the isotropic case a hexagonal grid is stable and all 6 triple junctions are in equilibrium. This isn't the case when the energies are not the same. We arrange the grains so that the boundaries around one grain would be high and the others would be low. We would expect the grain with the high energy boundaries to shrink but instead they grew as seen in Figure 2.2. This confirms that the free energy of the system is increasing. A discussion about this will be in the next section.

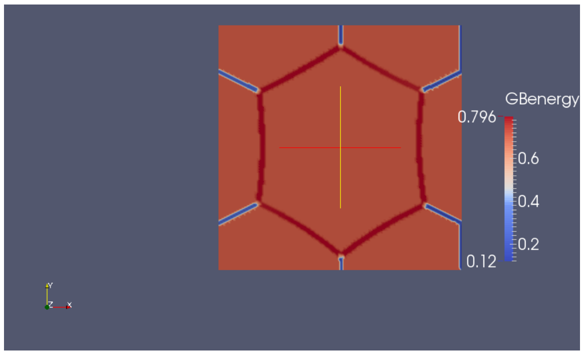


Figure 2.2 A hexagon grain showing that the high energy grain boundaries grow. This has to be because the triple junctions are treated incorrectly.

Chapter 3

User Object Implementation

As discussed in the previous section, single grains exhibited the expected behavior, while polycrystalline grain structures didn't. The high energy boundaries preferentially grew instead of preferentially shrinking as expected. This causes the free energy to increase in some systems. As we know from thermodynamics the free energy in an isolated system should always decrease.

3.1 Procedures

The difference between the two grain system and the many grain system is that there are triple junctions in the many grain system. This means that the triple junctions are incorrectly modeled. After much examination, I realized that this is because in the Allen-Cahn equation we don't include how the free energy will change as the grain boundary energy changes. In other words, the grain boundaries will move to a lower energy but when the grain boundary energy is recalculated the free energy actually increases. An extra term needed to be included. To figure out what the extra term would be I recalculated the functional derivative. The functional derivative is given by $\frac{\delta F}{\delta \eta_k} = \frac{\partial f}{\partial \eta_k} - \nabla \cdot \frac{\partial f}{\partial \nabla \eta_k}$. In the case of grain growth we have

$$f = \mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4} - \frac{\eta_i^2}{2}\Big) + \gamma \sum_{i=1}^{p}\sum_{j>i}^{p} \eta_i^2\eta_j^2 + \frac{1}{4}\Big) + \frac{\kappa}{2}\sum_{i=1}^{p}(\nabla \eta_i)^2.$$

Since both μ and κ are linearly dependent on σ we can rewrite the equation above as

$$f = \sigma\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4} - \frac{\eta_i^2}{2}\Big) + \gamma \sum_{i=1}^{p}\sum_{j>i}^{p} \eta_i^2\eta_j^2 + \frac{1}{4}\Big) + \frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla \eta_i)^2\Big),$$

where $C_\mu = \frac{\mu}{\sigma}$ and $C_\kappa = \frac{\kappa}{\sigma}$ both of which are constants.

$$\frac{\partial f}{\partial \eta_k} = \frac{\partial \sigma}{\partial \eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4} - \frac{\eta_i^2}{2}\Big) + \gamma \sum_{i=1}^{p}\sum_{j>i}^{p} \eta_i^2\eta_j^2 + \frac{1}{4}\Big) + \frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla \eta_i)^2\Big) + \mu(\eta_k^3 - \eta_k + 2\gamma\eta_k\sum_{j\neq k}^{p}\eta_j^2)$$

The only η dependence in σ comes from the weighting function $\sigma = \frac{\sum_{i=1}^{p}\sum_{j>i}^{p}\sigma_{ij}\eta_i^2\eta_j^2}{\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2}$. From this we obtain

$$\frac{\partial \sigma}{\partial \eta_k} = \frac{2\eta_k\sum_{j\neq k}^{p}\sigma_{kj}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2 - 2\eta_k\sum_{j\neq k}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\sigma_{ij}\eta_i^2\eta_j^2}{(\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2)^2}.$$

This derivative is only non-zero when the energy is anisotropic and there are 3 or more non-zero order parameters at that point. First, to illustrate that this term is always zero in the isotropic case, we replace the σ_{ij} with σ since all of them are the same. Doing this and pulling the σ out of the sum we obtain

$$\frac{\partial \sigma}{\partial \eta_k} = \frac{2\sigma\eta_k\sum_{j\neq k}^{p}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2 - 2\sigma\eta_k\sum_{j\neq k}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2}{(\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2)^2} = 0.$$

To show that there has to be 3 or more non-zero grains we will assume there are 2 non-zero order parameters, η_1 and η_2, and the rest are zero. The derivative after evaluating the sums becomes

$$\frac{\partial \sigma}{\partial \eta_1} = \frac{2\sigma_{12}\eta_1^3\eta_2^4 - 2\sigma_{12}\eta_1^3\eta_2^4}{\eta_1^4\eta_2^4} = 0$$

This goes along with my observation that the free energy problem is because of incorrect treatment at triple junctions, since this term is only non-zero at multi-junctions.

The other term in the functional derivative is

$$\nabla\cdot\frac{\partial f}{\partial\nabla\eta_k}=\nabla\cdot\frac{\partial\sigma}{\partial\nabla\eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4}-\frac{\eta_i^2}{2}\Big)+\gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2+\frac{1}{4}\Big)+\frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2\Big)+\nabla\cdot(\kappa\nabla\eta_k).$$

Since σ doesn't have an easily-defined dependence on $\nabla\eta$ we will not include the first term on the right hand side at this time. This term would rotate a boundary to a lower energy inclination. This probably isn't a very dominant effect, so we are not including it and hoping that we will still obtain accurate results.

For the functional derivative we have

$$\frac{\delta F}{\delta\eta_k}=\frac{\partial\sigma}{\partial\eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4}-\frac{\eta_i^2}{2}\Big)+\gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2+\frac{1}{4}\Big)+\frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2\Big)+\mu(\eta_k^3-\eta_k+2\gamma\eta_k\sum_{j\neq k}^{p}\eta_j^2)$$
$$-\nabla\cdot(\kappa\nabla\eta_k).$$

The only extra term from the isotropic case is

$$\frac{\partial\sigma}{\partial\eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4}-\frac{\eta_i^2}{2}\Big)+\gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2+\frac{1}{4}\Big)+\frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2\Big),$$

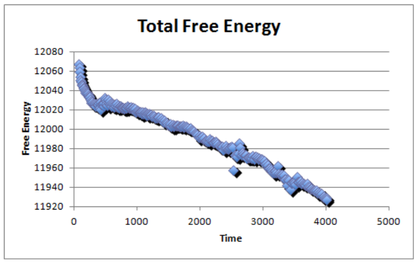
this term tells us how the free energy is changed because σ changes as η changes. Where

$$\frac{\partial\sigma}{\partial\eta_k}=\frac{2\eta_k\sum_{j\neq k}^{p}\sigma_{kj}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2-2\eta_k\sum_{j\neq k}\eta_j^2\sum_{i=1}^{p}\sum_{j>i}^{p}\sigma_{ij}\eta_i^2\eta_j^2}{(\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2)^2}.$$

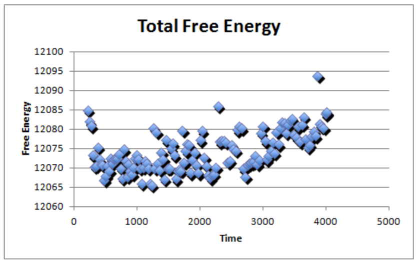
To insure that this term would cause the free energy to decrease I wrote a kernel that includes this term. I incorporated this kernel with the 5 DoF model. This would involved moving the calculations into a user object because auxiliary kernels could only pass back one number per point. Here we need to pass the energy of all of the pairs of active grains, so that we could evaluate $\frac{\partial\sigma}{\partial\eta_k}$. Previously, we do the weighting inside of the axillary kernel. To calculate the sums such as $\sum_{i=1}^{p}\sum_{j>i}^{p}\sigma_{ij}\eta_i^2\eta_j^2$ I only summed over the active grain pairs. This is valid since the non-active order parameters are very close or equal to zero. We do the weighting in the user object and it can pass the average energy or all the energies with the associated pair of grain boundary energies. The material only uses the averaged energy, but the kernel uses the map of energies.

3.2 Results

The free energy does decrease in the hexagonal test case as seen in Figure 3.1. The high energy grain shrinks. The Free energies of the arrangement with the kernel is seen in Figure 3.1a and without the kernel in Figure 3.1b.



(a) The free energy of a hexagonal grid with the new kernel. The Free energy is steadily decreasing.



(b) The free energy of a hexagonal grid without the new kernel. The free Energy is sporadic but has an upward trend.

Figure 3.1 The free energy of a hexagonal case with and without the new kernel.

One peculiar result that needs to be resolved is that when the ratio between the energies of the boundaries in a triple junction is high enough the low energy boundaries creep up the high energy boundary to create a lower energy triple junction. This isn't physical. This is a lot more noticeable when the ratio between high energy and low energy boundaries is high as seen in Figure 3.2 and Figure 3.3. A discussion of one way to fix this problem is in the conclusion.

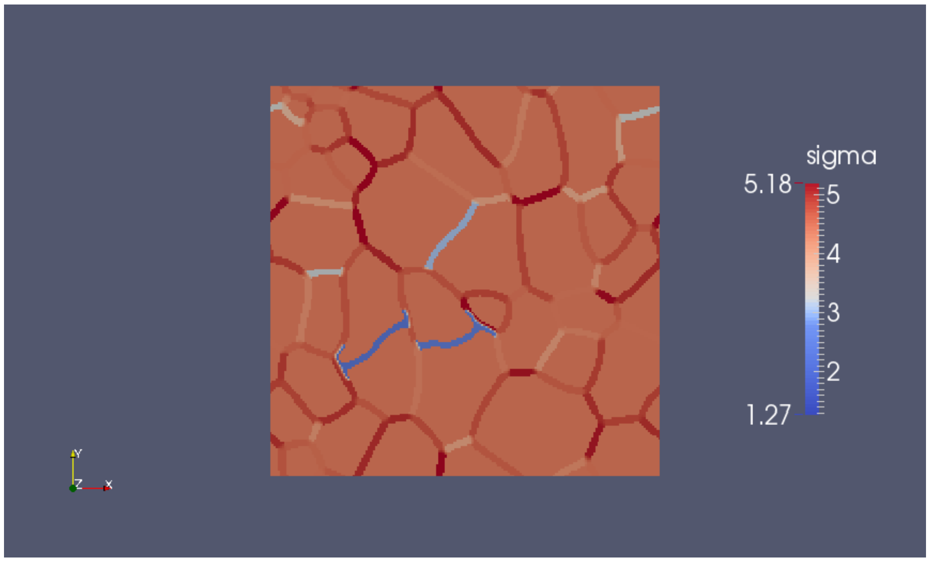


Figure 3.2 The grain boundary energy of a polycrystalline region. Low energy boundaries(blue) are creeping up high energy boundaries(red).

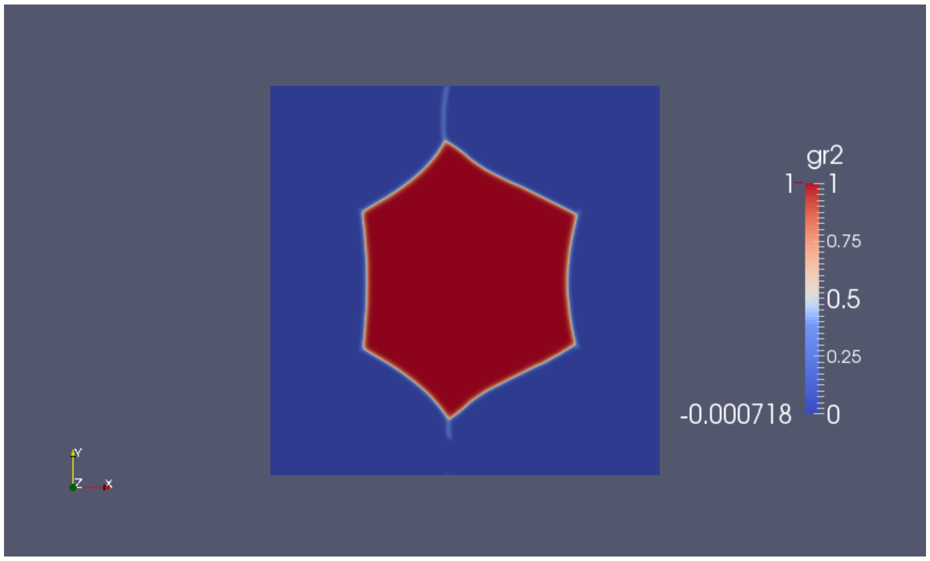


Figure 3.3 A hexagon grain with the two lower energies on the top connected to a high energy boundary. This figure shows the value of one of the order parameters. The grain growth in this case is proceeded by the creeping, and the creeping continues after the grain growth stopped.

Chapter 4

Conclusion

The goal of this project was to more accurately model grain growth by introducing anisotropic grain boundary energy. We used a 5 degree of freedom model to calculate the energy. There were a few phenomenon that we were looking for that separates isotropic grain growth and anisotropic grain growth. One of these is that high energy boundaries shrink preferentially. We saw this after we added another term to the Allen-Cahn equation. This showed that it is a force on the triple junction that causes this phenomenon. Another phenomenon that we are looking for is the formation of a texture. This is where grains of similar orientation are near each other(3). This affects material properties significantly. Another project being performed to look at this.

This study is being performed to compare quantitatively these simulations with experimental result. INL has X-ray diffraction data of the grain structure of UO_2 from before the reactor and after the fuel was used, that can be used to do this comparison. The comparison would compare grain boundary character distributions. A grain boundary character distribution tells us the distribution of twist and tilt boundaries and how far apart they are. This research showed the importance of triple junction kinetics on the results of grain growth simulations. We know that the free energy at a triple junction is higher. We could add a term to the free energy density to increase the free energy at triple junctions. This could solve the problem of the grain boundaries creeping up other boundaries. This would require reevaluating the functional derivative and adding the appropriate kernel. One possible improvement is to include some other observed phenomenon related to triple junctions such as triple junction drag (8).

Appendix A

Weak Form

Many differential equations can be written in either a differential form or an integral form. The differential form is also called the strong form, and the integral form is also called the weak form. One example where both examples are familiar is Gauss's law. The strong form is given by $\nabla\cdot\vec{E} = \frac{\rho}{\epsilon_0}$ to get to the weak form we could integrate over a volume obtaining $\int_R \nabla\cdot\vec{E}dV = \int_R \frac{\rho}{\epsilon_0}dV$. After using the divergence theorem and evaluating the right hand side we obtain the familiar result $\int_{\partial R}\vec{E}\cdot d\vec{A} = \frac{Q}{\epsilon_0}$.

To evaluate the Allen-Cahn equation in MOOSE we need it to be in the weak form. The strong form is

$$\frac{\partial\eta_k}{\partial t} = -L\Bigg[\frac{\partial\sigma}{\partial\eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4}-\frac{\eta_i^2}{2}\Big)+\gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2+\frac{1}{4}\Big)+\frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2\Big)$$

$$+\mu(\eta_k^3-\eta_k+2\gamma\eta_k\sum_{j\neq k}^{p}\eta_j^2)-\nabla\cdot(\kappa\nabla\eta_k)\Bigg].$$

Integrate both sides and apply the divergence theorem to obtain the weak form:

$$\int_R\frac{\partial\eta_k}{\partial t}dV = -L\int_R\frac{\partial\sigma}{\partial\eta_k}\Big(C_\mu\Big(\sum_{i=1}^{p}\Big(\frac{\eta_i^4}{4}-\frac{\eta_i^2}{2}\Big)+\gamma\sum_{i=1}^{p}\sum_{j>i}^{p}\eta_i^2\eta_j^2+\frac{1}{4}\Big)+\frac{C_\kappa}{2}\sum_{i=1}^{p}(\nabla\eta_i)^2\Big)dV$$

$$-L\int_R\mu(\eta_k^3-\eta_k+2\gamma\eta_k\sum_{j\neq k}^{p}\eta_j^2)dV-L\int_{\partial R}(\kappa\nabla\eta_k)\cdot d\vec{A}.$$

Appendix B

Code

B.1 AnisoGBEnergyUserObject.C

```
#include "AnisoGBEnergyUserObject.h"

template<>
InputParameters validParams<AnisoGBEnergyUserObject>()
{
  InputParameters params = validParams<ElementUserObject>();
  params.addClassDescription("Calculates the Grain Boundary energy using
  Bulatov's method");
  params.addRequiredCoupledVarWithAutoBuild("v", "var_name_base", "op_num",
  "Array of coupled variables");
  params.addRequiredParam<UserObjectName>("euler_angle_provider", "Name of the
  euler angle provider user object");
  MooseEnum materials("Cu Al Au Ni UO2", "Cu");
  params.addParam<MooseEnum>("material", materials, "Which material parameters to use");
  params.addRequiredParam<Real>("GBenergy_iso","The average grain boundary energy");
  params.addRequiredParam<unsigned int>("grain_num", "number of grains");
  params.addRequiredParam<UserObjectName>("GrainTracker_object", "The GrainTracker
  UserObject to get values from.");

  return params;
}

AnisoGBEnergyUserObject::AnisoGBEnergyUserObject(const InputParameters & parameters) :
    ElementUserObject(parameters),
    _ncrys(getParam<unsigned int>("grain_num")),
    _nop(coupledComponents("v")),
    _mesh_changed(true),
    _GBenergy_iso(getParam<Real>("GBenergy_iso")),
    _material(getParam<MooseEnum>("material")),
    _euler(getUserObject<EulerAngleProvider>("euler_angle_provider")),
    _grain_tracker(getUserObject<GrainTracker>("GrainTracker_object")),
    _par_vec(43),
    _geom100(4),
    _geom110(4),
    _geom111(4),
    _axes100(3),
    _dirs100(3),
    _axes110(6),
    _dirs110(6),
    _axes111(4),
    _dirs111(4),
    _rotX90(1, 0, 0, 0, 0, 1, 0, -1, 0),   //Rotation by +90 degrees around X axis.
    _rotY90(0, 0, -1, 0, 1, 0, 1, 0, 0),   //Rotation by +90 degrees around Y axis
    _rotZ90(0, 1, 0, -1, 0, 0, 0, 0, 1),   //Rotation by +90 degrees around Z axis
    _rotZ90m(0, -1, 0, 1, 0, 0, 0, 0, 1)
{
  //set the vectors that don't change
  makeParVec();
  set_axes100();
  set_axes110();
  set_axes111();


  // Resize and initialize values
  _v.resize(_nop);
  _grad_v.resize(_nop);
  _orientation_matrix.resize(_ncrys);

  for (unsigned int ops = 0 ; ops < _nop; ++ops)
  {
    _v[ops] = &coupledValue("v", ops);
    _grad_v[ops] = &coupledGradient("v", ops);
  }

  if (_euler.getGrainNum() < _ncrys)
    mooseError("Euler angle provider has too few angles.");

  for (unsigned int crys = 0; crys < _ncrys; ++crys)
  {
    RealVectorValue Euler_Angles = _euler.getEulerAngles(crys);
    _orientation_matrix[crys] = RotationTensor(Euler_Angles);
  }
}

void
AnisoGBEnergyUserObject::initialize()
{
  if (_mesh_changed)
  {
    _GBEnergies.clear();
    _rebuild_map = true;
  }
  else
    _rebuild_map = false;

  _mesh_changed = false;
}

void
AnisoGBEnergyUserObject::execute()
{
  if (_rebuild_map)
  {
    RealTensorValue _rotation_matrix; // rotates _gbdir to [100]
    RealVectorValue _gbdir; // the normal to the grain boundary

    // Get list of active order parameters from grain tracker
    const std::vector<std::pair<unsigned int, unsigned int> > & active_ops = _
    grain_tracker.getElementalValues(_current_elem->id());

    unsigned int n_active_ops = active_ops.size();
    if (n_active_ops < 1 && _t_step > 0 )
    mooseError("No active order parameters");

    Real GBen_pair;
    RealVectorValue axis100(1, 0, 0);
    for (int op1 = 0; op1 < n_active_ops; ++op1)
    {
      for(int op2 = op1 + 1; op2 < n_active_ops; ++op2)
      {
        _gbdir = (*_grad_v[active_ops[op1].second])[0] /
        (*_grad_v[active_ops[op1].second])[0].size();
        _rotation_matrix = RotationMatrix::rotVec1ToVec2(_gbdir, axis100);
        //Use Bulatov's function
        GBen_pair = gB5DOF(_rotation_matrix * _orientation_matrix[(active_ops[op1].first)],
        _rotation_matrix * _orientation_matrix[(active_ops[op2].first)]);
        std::pair<unsigned int, unsigned int> op_pair(active_ops[op1].second,
                active_ops[op2].second);
        _GBEnergies[_current_elem->id()].insert(std::pair<std::pair<unsigned int,
        unsigned int>, Real>(op_pair, GBen_pair));
      }
    }
  }
  return;
}

void AnisoGBEnergyUserObject::meshChanged()
{
  _mesh_changed = true;
}

void
AnisoGBEnergyUserObject::threadJoin(const UserObject & y)
{
  if (_rebuild_map)
  {
    const AnisoGBEnergyUserObject & uo = static_cast<const AnisoGBEnergyUserObject &>(y);
    _GBEnergies.insert(uo._GBEnergies.begin(), uo._GBEnergies.end());
  }
}

const std::map< std::pair< unsigned int , unsigned int>, Real> &
AnisoGBEnergyUserObject::getGBEnergies(unsigned int elem_id) const
{
  if (_GBEnergies.count(elem_id))
    return _GBEnergies.find(elem_id)->second;
  else
    return _empty_map; //It needs this before the map is filled
}

Real
AnisoGBEnergyUserObject::getWeightedEnergy(unsigned int elem_id, unsigned int qp) const
{
  Real _GBenergy = _GBenergy_iso;
  const std::map<std::pair<unsigned int, unsigned int>, Real> & gb_energies =
  _GBEnergies.find(elem_id)->second;
  //if we are in a grain or don't have data for the element use the isotropic energy
  if (!gb_energies.empty() && _GBEnergies.count(elem_id))
  {
    typedef std::map<std::pair<unsigned int,unsigned int>,Real>::const_iterator energies_it;
    Real SumEtaij = 0;
    Real SumEtaSigmaij = 0;
    for (energies_it it = gb_energies.begin(); it != gb_energies.end(); ++it)
    {
      unsigned int op1 = (it->first).first;
      unsigned int op2 = (it->first).second;
      Real sigmaij = it->second;
      SumEtaij += (*_v[op1])[qp] * (*_v[op1])[qp] * (*_v[op2])[qp] * (*_v[op2])[qp];
      SumEtaSigmaij += sigmaij * (*_v[op1])[qp] * (*_v[op1])[qp] *
      (*_v[op2])[qp] * (*_v[op2])[qp];
    }
    _GBenergy = SumEtaSigmaij / SumEtaij;
  }
  return _GBenergy;
}

void
AnisoGBEnergyUserObject::makeParVec()
{
  if (_material == "UO2")
  {
    const Real temp[43] = {1.615789, 0.405, 7.39, 0.352, 26.2, 0.65, 6.1, 0.363719,
       0.000141, 0.635392, 1.320973, 34.1, 1.030398, 0.876971, 1.038988, 0.893062,
       1.041893, 0.312093, 0.739168, 0.154383, 0.905391, 0.422022, 1.211934, 1.092599,
       1.20006, 1.0, 1.238721, 0.6127202, 0.576808, 1.193228, 0.944648, 2.09343,
       2.717258, 0.606289, 1.97468, 0.546169, 0.832776, 0.164954, 1.146105, 1.159805,
       1.330302, 3.730652};

    for (int i = 0; i < 43; ++i)
      _par_vec[i] = temp[i];
  }
  else
  {
    // _par_vec can be defied for just Al and Cu and then interpolated for the
    // other FCC metals then the energy scaling parameter, _par_vec[0], defined seperately
    const Real par42Al[42] = {0.405204179289160, 0.738862004021890, 0.351631012630026,
      2.40065811939667, 1.34694439281655, 0.352260396651516, 0.602137375062785,
      1.58082498976078, 0.596442399566661, 1.30981422643602, 3.21443408257354,
      0.893016409093743, 0.835332505166333, 0.933176738717594, 0.896076948651935,
      0.775053293192055, 0.391719619979054, 0.782601780600192, 0.678572601273508,
      1.14716256515278, 0.529386201144101, 0.909044736601838, 0.664018011430602,
      0.597206897283586, 0.200371750006251, 0.826325891814124, 0.111228512469435,
      0.664039563157148, 0.241537262980083, 0.736315075146365, 0.514591177241156,
      1.73804335876546, 3.04687038671309, 1.48989831680317, 0.664965104218438,
      0.495035051289975, 0.495402996460658, 0.468878130180681, 0.836548944799803,
      0.619285521065571, 0.844685390948170, 1.02295427618256};
    const Real par42Cu[42] = {0.405204179289160, 0.738862004021890, 0.351631012630026,
      2.40065811939667, 1.34694439281655, 3.37892632736175, 0.602137375062785,
      1.58082498976078, 0.710489498577995, 0.737834049784765, 3.21443408257354,
      0.893016409093743, 0.835332505166333, 0.933176738717594, 0.896076948651935,
      0.775053293192055, 0.509781056492307, 0.782601780600192, 0.762160812499734,
      1.10473084066580, 0.529386201144101, 0.909044736601838, 0.664018011430602,
      0.597206897283586, 0.200371750006251, 0.826325891814124, 0.0226010533470218,
      0.664039563157148, 0.297920289861751, 0.666383447163744, 0.514591177241156,
      1.73804335876546, 2.69805148576400, 1.95956771207484, 0.948894352912787,
      0.495035051289975, 0.301975031994664, 0.574050577702240, 0.836548944799803,
      0.619285521065571, 0.844685390948170, 0.0491040633104212};
    Real AlCuParameter;
    Real eRGB;
    if (_material == "Ni")
    {
      eRGB = 1.44532834613925;
      AlCuParameter = 0.767911805073948;
    }
    else if (_material =="Al")
    {
      eRGB = 0.547128733614891;
      AlCuParameter = 0;
    }
    else if (_material == "Au")
    {
      eRGB = 0.529912885175204;
      AlCuParameter = 0.784289766313152;
    }
    else if (_material == "Cu")
    {
      eRGB = 1.03669431227427;
      AlCuParameter = 1;
    }
    _par_vec[0] = eRGB;
    for (unsigned int i = 1; i < 43; ++i)
      _par_vec[i] = par42Al[i - 1] + AlCuParameter * (par42Cu[i - 1] - par42Al[i - 1]);
  }
}

void
AnisoGBEnergyUserObject::set_axes100()
{
  //define 100 axes, normalize
  _axes100[0](0) = 1;
  _axes100[0](1) = 0;
  _axes100[0](2) = 0;

  _axes100[1](0) = 0;
  _axes100[1](1) = 1;
  _axes100[1](2) = 0;

  _axes100[2](0) = 0;
  _axes100[2](1) = 0;
  _axes100[2](2) = 1;
  //define crystal direction
  _dirs100[0](0) = 0;
  _dirs100[0](1) = 1;
  _dirs100[0](2) = 0;

  _dirs100[1](0) = 0;
  _dirs100[1](1) = 0;
  _dirs100[1](2) = 1;

  _dirs100[2](0) = 1;
  _dirs100[2](1) = 0;
  _dirs100[2](2) = 0;
}

void
AnisoGBEnergyUserObject::set_axes110()
{
  const Real s2 = 1/std::sqrt(2);
  //define the 110 axes, normalize
  _axes110[0](0) = s2;
  _axes110[0](1) = s2;
  _axes110[0](2) = 0;

  _axes110[1](0) = s2;
  _axes110[1](1) = -s2;
  _axes110[1](2) = 0;

  _axes110[2](0) = s2;
  _axes110[2](1) = 0;
  _axes110[2](2) = s2;

  _axes110[3](0) = s2;
  _axes110[3](1) = 0;
  _axes110[3](2) = -s2;

  _axes110[4](0) = 0;
  _axes110[4](1) = s2;
  _axes110[4](2) = s2;

  _axes110[5](0) = 0;
  _axes110[5](1) = s2;
  _axes110[5](2) = -s2;
  //define the crystal direction perpendicular to each rotation axis.
  //The formalism demands that this be an axis of at least two-fold symmetry.
  _dirs110[0](0) = 0;
  _dirs110[0](1) = 0;
  _dirs110[0](2) = 1;

  _dirs110[1](0) = 0;
  _dirs110[1](1) = 0;
  _dirs110[1](2) = 1;

  _dirs110[2](0) = 0;
  _dirs110[2](1) = 1;
  _dirs110[2](2) = 0;

  _dirs110[3](0) = 0;
  _dirs110[3](1) = 1;
  _dirs110[3](2) = 0;

  _dirs110[4](0) = 1;
  _dirs110[4](1) = 0;
  _dirs110[4](2) = 0;

  _dirs110[5](0) = 1;
  _dirs110[5](1) = 0;
  _dirs110[5](2) = 0;
}

void
AnisoGBEnergyUserObject::set_axes111()
{
  const Real s3 = 1/std::sqrt(3);
  const Real s2 = 1/std::sqrt(2);
  //define the 111 axes, normalize.
  _axes111[0](0) = s3;
  _axes111[0](1) = s3;
  _axes111[0](2) = s3;

  _axes111[1](0) = s3;
  _axes111[1](1) = -s3;
  _axes111[1](2) = -s3;

  _axes111[2](0) = -s3;
  _axes111[2](1) = s3;
  _axes111[2](2) = -s3;

  _axes111[3](0) = -s3;
  _axes111[3](1) = -s3;
  _axes111[3](2) = s3;
  //define crystal direction
  _dirs111[0](0) = s2;
  _dirs111[0](1) = -s2;
  _dirs111[0](2) = 0;

  _dirs111[1](0) = s2;
  _dirs111[1](1) = s2;
  _dirs111[1](2) = 0;

  _dirs111[2](0) = s2;
  _dirs111[2](1) = s2;
  _dirs111[2](2) = 0;

  _dirs111[3](0) = s2;
  _dirs111[3](1) = -s2;
  _dirs111[3](2) = 0;
}
```

```
Real
AnisoGBEnergyUserObject::gB5DOF(RealTensorValue P, RealTensorValue S)
{
  // P and S are the rotation Matrices for the two grains
  distancesToSet(P, S, _geom100, _axes100, _dirs100);
  distancesToSet(P, S, _geom110, _axes110, _dirs110);
  distancesToSet(P, S, _geom111, _axes111, _dirs111);

  return weightedMeanEnergy();
}

void
AnisoGBEnergyUserObject::distancesToSet(const RealTensorValue & P, RealTensorValue & S,
  std::vector<std::vector<Real> > & geom, const std::vector<RealVectorValue> & axes,
  const std::vector<RealVectorValue> & dirs)
{
  unsigned int naxes = axes.size();

  Real dismax = 0.999999;  //force the distance to be strictly less than one,
  //determine the number of axes for a given rotation axis

  Real period = libMesh::pi*naxes/6;

  /* Create 24 symmetry equivalent variants of S
  This is the coset appropriate for the rotation convention where S*P
  is the misorientation represented in the grain frame.  If you're
  getting odd results, e.g. misorientations that you know are CSL are
  coming out entirely wrong, you may be using the opposite convention;
  try replacing P and S with P' and S'.*/
  std::vector< RealTensorValue > V(24); //initialize 24x3x3 array to hold variants of Q
  V[0] = S;  //input S into slot 0 of array

  RealTensorValue Temp = S;

  for (unsigned int i = 1; i < 4; i++)   //Rotate three times around X by +90 degrees
  {
    Temp = Temp * _rotX90;
    V[i] = Temp;
  }
  for (unsigned int i = 4; i < 16; i++)   //Rotate three times around Y by +90 degrees
  {
    Temp = V[i-4];
    Temp = Temp * _rotY90;
    V[i] = Temp;
  }
  for (unsigned int i = 16; i < 20; i++)   //Rotate around Z by +90 degrees
  {
    Temp = V[i-16];
    Temp = Temp * _rotZ90;
    V[i] = Temp;
  }
  for (unsigned int i = 20; i < 24; i++)   //Rotate around Z by -90 degrees
  {
    Temp = V[i-20];
    Temp = Temp * _rotZ90m;
    V[i] = Temp;
  }
  //Preallocate all parameter lists at their maximum possible sizes
  //Redundant representations will be removed at the end
  geom[0].resize(24*naxes); // distances
  geom[1].resize(24*naxes); // ksis
  geom[2].resize(24*naxes); // etas
  geom[3].resize(24*naxes); // phis

  unsigned int thisindex = 0;   //Number of hits found so far
```

```
//Step through all combinations of symmetrically-equivalent axes and coset elements V
for (unsigned int i = 0; i < naxes; i++)
{

  //Completing the orthonormal coordinate set.
  //theta1 and theta2 are defined in the plane spanned by (dir,dir2)
  RealVectorValue dir2 = axes[i].cross(dirs[i]);

  RealTensorValue R;

  //for each symmetry-related variant of the second grain
  for (unsigned int j = 0; j < 24; j++)
  {
    R = V[j].transpose() * P;
    //This rotates any vector in cube P into a vector in cube S

    std::vector<Real> q(4,0);

    mat2Quat(R,q);   //Calculation from here on out is much easier with quaternions.

    Real lq = std::sqrt(q[1] * q[1] + q[2] * q[2] + q[3] * q[3]);
    RealVectorValue axi(q[1] / lq,  q[2] / lq, q[3] / lq);


    Real psi = 2*std::acos(q[0]);   //Rotation angle

    Real dot_p = axi * axes[i];

    //Compute rotational distance from boundary P/S to the rotation set i.
    //This formula produces 2*sin(delta/2), where delta is the angle of closest approach.
    Real dis = 2 * std::sqrt(std::abs(1 - dot_p * dot_p)) * std::sin(psi / 2);

    if (dis < dismax)
    {
      //angle of rotation about ax that most closely approximates R
      Real theta = 2 * std::atan(dot_p * std::tan(psi / 2));

      //Compute the normal of the best-fitting GB in grain 1
      RealVectorValue n1(P(0,0), P(0,1), P(0,2));

      RealVectorValue n2(V[j](0,0), V[j](0,1), V[j](0,2));

      RealTensorValue RA;
      std::vector<Real> q2(4);
      q2[0] = std::cos(theta / 2);
      q2[1] = std::sin(theta / 2) * axes[i](0);
      q2[2] = std::sin(theta / 2) * axes[i](1);
      q2[3] = std::sin(theta / 2) * axes[i](2);
      quat2Mat(q2,RA);   //Rotation matrix that most closely approximates R

      //from now on we're dealing with the idealized rotation RA, not the original R
      RealVectorValue m1 = n1 + RA.transpose()*n2;

      if (m1.size() >= 0.000001)   //discard values for very large distances
      {

        //halfway between the two normal vectors from the two grains
        Real l = m1.size();
        m1 /= l;

        //same representation in the other grain
        RealVectorValue m2 = RA * m1;

        //compute the inclination angle for the common rotation axis
        Real phi = std::acos(std::abs(m1 * axes[i]));
```

```
          Real theta1;
          Real theta2;

          //partition the total rotation angle "theta"
          if (std::abs(axes[i] * m1) > 0.9999)   //check if best-fitting GB is pure twist
          {
            theta1 = -theta / 2;   //eta is meaningless for a twist boundary
            theta2 = theta / 2;
          }
          else
          {
            //Project m1 and m2 into the plane normal to ax and determine
            //the rotation angles of them relative to dir
            theta1 = std::atan2(dir2 * m1, dirs[i] * m1);
            theta2 = std::atan2(dir2 * m2, dirs[i] * m2);
          }

          //Reduce both angles to interval (-period/2,period/2)
          //semi-open with a small numerical error
          theta2 = theta2 - std::round(theta2 / period) * period;
          theta1 = theta1 - std::round(theta1 / period) * period;

          //implement the semi-open interval in order to avoid an annoying
          //numerical problem where certain representations are Real-counted
          if (std::abs(theta2 + period / 2) < 0.000001)
            theta2 = theta2 + period;
          if (std::abs(theta1 + period / 2) < 0.000001)
            theta1 = theta1 + period;

          /* Since this is only being run on fcc and fluorite elements, which are
          centrosymmetric, and all dir vectors are 2-fold axes, then
          the operations of swapping theta1 and theta2, and of
          multilying both by -1, are symmetries for the energy
          function. This lets us fold everything into a small right
          triangle in (ksi,eta) space: */
          Real ksi = std::abs(theta2 - theta1);
          Real eta = std::abs(theta2 + theta1);

          //round everything to 1e-6, so that negligible numerical differences are dropped
          geom[0][thisindex] = 0.000001 * std::round(dis * 1000000);
          geom[1][thisindex] = 0.000001 * std::round(ksi * 1000000);
          geom[2][thisindex] = 0.000001 * std::round(eta * 1000000);
          geom[3][thisindex] = 0.000001 * std::round(phi * 1000000);
          thisindex = thisindex + 1;
        }
        else
        {
          //discard large geom[0]
          geom[0].erase(geom[0].begin() + thisindex);
          geom[1].erase(geom[1].begin() + thisindex);
          geom[2].erase(geom[2].begin() + thisindex);
          geom[3].erase(geom[3].begin() + thisindex);
        }
      }
      else
      {
        //dump excess preallocated slots
        geom[0].erase(geom[0].begin() + thisindex);
        geom[1].erase(geom[1].begin() + thisindex);
        geom[2].erase(geom[2].begin() + thisindex);
        geom[3].erase(geom[3].begin() + thisindex);
      }
    }
  }
  sortGeom(geom);
```

```
}

void
AnisoGBEnergyUserObject::sortGeom(std::vector< std::vector<Real> > & geom)
{
  //sort values by distance, ksi, eta, and phi
  bool change = true;
  while (change)
  {
    change = false;
    for (unsigned int i = 0; i < (geom[0].size() - 1); ++i)
    {
      if (geom[0][i+1] < geom[0][i])
      {
        Real temp = geom[0][i];
        geom[0][i] = geom[0][i+1];
        geom[0][i+1] = temp;
        temp = geom[1][i];
        geom[1][i] = geom[1][i+1];
        geom[1][i+1] = temp;
        temp = geom[2][i];
        geom[2][i] = geom[2][i+1];
        geom[2][i+1] = temp;
        temp = geom[3][i];
        geom[3][i] = geom[3][i+1];
        geom[3][i+1] = temp;
        change = true;
      }
      else if (geom[0][i+1] == geom[0][i])
      {
        if (geom[1][i+1] < geom[1][i])
        {
          Real temp = geom[0][i];
          geom[0][i] = geom[0][i+1];
          geom[0][i+1] = temp;
          temp = geom[1][i];
          geom[1][i] = geom[1][i+1];
          geom[1][i+1] = temp;
          temp = geom[2][i];
          geom[2][i] = geom[2][i+1];
          geom[2][i+1] = temp;
          temp = geom[3][i];
          geom[3][i] = geom[3][i+1];
          geom[3][i+1] = temp;
          change = true;
        }
        else if (geom[1][i+1] == geom[1][i])
        {
          if (geom[2][i+1] < geom[2][i])
          {
            Real temp = geom[0][i];
            geom[0][i] = geom[0][i+1];
            geom[0][i+1] = temp;
            temp = geom[1][i];
            geom[1][i] = geom[1][i+1];
            geom[1][i+1] = temp;
            temp = geom[2][i];
            geom[2][i] = geom[2][i+1];
            geom[2][i+1] = temp;
            temp = geom[3][i];
            geom[3][i] = geom[3][i+1];
            geom[3][i+1] = temp;
            change = true;
          }
          else if (geom[2][i+1] == geom[2][i])
          {
            if (geom[3][i+1] < geom[3][i])
            {
              Real temp = geom[0][i];
              geom[0][i] = geom[0][i+1];
              geom[0][i+1] = temp;
              temp = geom[1][i];
              geom[1][i] = geom[1][i+1];
              geom[1][i+1] = temp;
              temp = geom[2][i];
              geom[2][i] = geom[2][i+1];
              geom[2][i+1] = temp;
              temp = geom[3][i];
              geom[3][i] = geom[3][i+1];
              geom[3][i+1] = temp;
              change = true;
            }
          }
        }
      }
    }
  }
  //remove duplicate values. Real-counting in the same representation of one
  //boundary messes up the
  //weighing functions in weightedMeanEnergy()
  for (unsigned int i = 0; i < (geom[0].size()-1); ++i)
  {
    for (unsigned int j = i+1; j < geom[0].size(); ++j)
    {
      if (geom[0][i] == geom[0][j] && geom[1][i] == geom[1][j] &&
            geom[2][i] == geom[2][j] && geom[3][i] == geom[3][j])
      {
        geom[0].erase(geom[0].begin() + j);
        geom[1].erase(geom[1].begin() + j);
        geom[2].erase(geom[2].begin() + j);
        geom[3].erase(geom[3].begin() + j);
        j--;
      }
    }
  }
  //remove excess zeroes
  for (unsigned int i = 0; i < geom[0].size(); ++i)
  {
    if (geom[0][i]== 0 && geom[1][i] == 0 && geom[2][i] == 0 && geom[3][i] == 0)
    {
      geom[0].erase(geom[0].begin() + i);
      geom[1].erase(geom[1].begin() + i);
      geom[2].erase(geom[2].begin() + i);
      geom[3].erase(geom[3].begin() + i);
    }
  }
}

void
AnisoGBEnergyUserObject::quat2Mat(const std::vector<Real> & q, RealTensorValue & _M)
{
  Real d = q[0] * q[0] + q[1] * q[1] + q[2] * q[2] + q[3] * q[3];
  _M(0, 0) = (q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] * q[3]) / d;
  _M(0, 1) = (2 * q[1] * q[2] - 2 * q[0] * q[3]) / d;
  _M(0, 2) = (2 * q[1] * q[3] + 2 * q[0] * q[2]) / d;
  _M(1, 0) = (2 * q[1] * q[2] + 2 * q[0] * q[3]) / d;
  _M(1, 1) = (q[0] * q[0] - q[1] * q[1] + q[2] * q[2] - q[3] * q[3]) / d;
  _M(1, 2) = (2 * q[2] * q[3] - 2 * q[0] * q[1]) / d;
  _M(2, 0) = (2 * q[1] * q[3] - 2 * q[0] * q[2]) / d;
  _M(2, 1) = (2 * q[2] * q[3] + 2 * q[0] * q[1]) / d;
  _M(2, 2) = (q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] * q[3]) / d;
}

void
AnisoGBEnergyUserObject::mat2Quat(const RealTensorValue & m, std::vector<Real> & q)
{
  Real t = m(0, 0) + m(1, 1) + m(2, 2);
  q[0] = std::sqrt(1 + t) / 2;

  if (t > -0.999999999)
  {
    q[1] = (m(1, 2) - m(2, 1)) / (4 * q[0]);
    q[2] = (m(2, 0) - m(0, 2)) / (4 * q[0]);
    q[3] = (m(0, 1) - m(1, 0)) / (4 * q[0]);
  }
  else
  {
    q[0] = 0;
    q[3] = std::sqrt(-(m(0, 0) + m(1, 1)) / 2);
    if (std::abs(q[3]) > 0.00000002)   //Check for singularity, allowing numerical error
    {
      q[1] = m(0, 2) / (2 * q[3]);
      q[2] = m(1, 2) / (2 * q[3]);
      q[3] = q[3];
    }
    else
    {
      q[1] = std::sqrt((m(0, 0) + 1) / 2);
      if (q[1] != 0)
      {
        q[1] = q[1];
        q[2] = m(1, 0) / (2 * q[1]);
        q[3] = 0;
      }
      else
      {
        q[1] = 0;
        q[2] = 1;
        q[3] = 0;
      }
    }
  }
  q[1] = -q[1];
  q[2] = -q[2];
  q[3] = -q[3];
}

Real
AnisoGBEnergyUserObject::weightedMeanEnergy()
{
  //Pull out the parameters relevant to the weighting of the 100, 110, and 111 sets
  const Real & eRGB = _par_vec[0];   //The only dimensioned parameter. The energy scale,
  //set by the energy of a random boundary.
  const Real & d0100 = _par_vec[1];  //Maximum distance for the 100 set. Also the distance
  //scale for the RSW weighting function.
  const Real & d0110 = _par_vec[2];  //Same for the 110 set
  const Real & d0111 = _par_vec[3];  //Same for the 111 set
  const Real & weight100 = _par_vec[4];  //Weight for the 100 set, relative to the
  //random boundary
  const Real & weight110 = _par_vec[5];  //Same for 110
  const Real & weight111 = _par_vec[6];  //Same for 111
```

```
Real offset = 0.00001;//Cutoff of weighting function at small d for numerical purposes

//Calculate energy lists in units of eRGB
std::vector<Real> e100(_geom100[0].size());
std::vector<Real> e110(_geom110[0].size());
std::vector<Real> e111(_geom111[0].size());

set100(e100);
set110(e110);
set111(e111);

std::vector<Real> s100(_geom100[0].size());
std::vector<Real> s110(_geom110[0].size());
std::vector<Real> s111(_geom111[0].size());

//Calculate the weights, in a manner designed to give an RSW-like function of distance
//Note it calculates a weight for every representation of the boundary in each set
for (unsigned int i = 0; i < _geom100[0].size(); i++)
{
  if (_geom100[0][i] > d0100)   //weight saturates at zero above d0
    s100[i] = 1;
  else if (_geom100[0][i] < (offset * d0100))//Avoid nan's, replace with something
  //small but finite
    s100[i] = offset * libMesh::pi / 2;
  else
    s100[i] = std::sin((libMesh::pi / 2) * _geom100[0][i] / d0100);
}

//same for 110
for (unsigned int i = 0; i < _geom110[0].size(); ++i)
{
  if (_geom110[0][i] > d0110)
    s110[i] = 1;
  else if (_geom110[0][i] < (offset * d0110))
    s110[i] = offset*libMesh::pi / 2;
  else
    s110[i] = std::sin((libMesh::pi / 2)*_geom110[0][i] / d0110);
}

//same for 111
for (unsigned int i = 0; i < _geom111[0].size(); i++)
{
  if (_geom111[0][i] > d0111)
    s111[i] = 1;
  else if (_geom111[0][i] < (offset * d0111))
    s111[i] = offset*libMesh::pi / 2;
  else
    s111[i] = std::sin((libMesh::pi / 2) * _geom111[0][i] / d0111);
}
Real w100;
Real w110;
Real w111;

Real sum100 = 0;
Real sum110 = 0;
Real sum111 = 0;

Real sumw100 = 0;
Real sumw110 = 0;
Real sumw111 = 0;

//calculate weights and sum up results
for (unsigned int i = 0; i < _geom100[0].size(); i++)
{
  w100 = (1 / (s100[i] * (1 - 0.5 * std::log(s100[i]))) - 1) * weight100;
```

```
    sum100 += (e100[i] * w100);
    sumw100 += w100;
  }
  for (unsigned int i = 0; i < _geom110[0].size(); i++)
  {
    w110 = (1/(s110[i] * (1-0.5*std::log(s110[i]))) - 1) * weight110;
    sum110 += (e110[i] * w110);
    sumw110 += w110;
  }
  for (unsigned int i = 0; i < _geom111[0].size(); ++i)
  {
    w111 = (1 / (s111[i] * (1 - 0.5 * std::log(s111[i]))) - 1) * weight111;
    sum111 += (e111[i] * w111);
    sumw111 += w111;
  }
  //calculate energy
  Real _en = eRGB * (sum100 + sum110 + sum111) / (sumw100 + sumw110 + sumw111);
  return _en;
}

void
AnisoGBEnergyUserObject::set100(std::vector<Real> & e100)
{
  const Real & pwr1 = _par_vec[7];  //100 tilt/twist mix power law: twist
  const Real & pwr2 = _par_vec[8];  //100 tilt/twist mix power law: tilt

  std::vector<Real> entwist(_geom100[0].size());
  std::vector<Real> entilt(_geom100[0].size());

  twist100(_geom100[1], entwist);
  aTGB100(_geom100[2], _geom100[1], entilt);

  for (unsigned int i = 0; i < _geom100[1].size(); i++)
    e100[i] = entwist[i] * std::pow((1 - (2 * _geom100[3][i] / libMesh::pi)), pwr1),
    + entilt[i] * std::pow((2 * _geom100[3][i] / libMesh::pi), pwr2);
}

void
AnisoGBEnergyUserObject::twist100(std::vector<Real> & ksi, std::vector<Real> & entwist)
{
  const Real & Emax = _par_vec[9];   //100 twist maximum energy
  const Real & a = _par_vec[10];  //100 twist RSW shape factor.
  Real period = libMesh::pi / 2;  //twist period

  for (unsigned int i = 0; i < ksi.size(); ++i)
  {
    ksi[i] = fmod(std::abs(ksi[i]), period);  //rotation symmetry
    if (ksi[i] > period / 2)
      ksi[i] = period-ksi[i];

    //implement an RSW function of ksi
    Real xlogx = std::sin(2 * ksi[i]) * std::log(std::sin(2 * ksi[i]));
    if (std::isnan(xlogx))  //force the limit to zero as x->0
      xlogx = 0;

    entwist[i] = Emax * (std::sin(2 * ksi[i]) - a * xlogx);
  }
}

void
AnisoGBEnergyUserObject::aTGB100(std::vector<Real> & eta, std::vector<Real> & ksi,
std::vector<Real> & entilt)
{
  const Real & pwr = _par_vec[11];  //100 aTGB interpolation power law
  Real period = libMesh::pi/2;
  std::vector<Real> en1(ksi.size());
  std::vector<Real> en2(ksi.size());
  std::vector<Real> temp(ksi.size());

  sTGB100(ksi, en1);   //value at eta = 0

  for (unsigned int i = 0; i < ksi.size(); i++)
    temp[i] = period - ksi[i];
  sTGB100(temp, en2);   //value at eta = pi/2

  //eta dependence is a power law that goes from the higher to the lower,
  //whichever direction that is
  for (unsigned int i = 0; i < ksi.size(); i++)
  {
    if (en1[i] >= en2[i])
      entilt[i] = en1[i] - (en1[i] - en2[i]) * std::pow((eta[i] / period), pwr);
    else
      entilt[i] = en2[i] - (en2[i] - en1[i]) * std::pow((1 - (eta[i] / period)), pwr);
  }
}

void
AnisoGBEnergyUserObject::sTGB100(std::vector<Real> & ksi, std::vector<Real> & _en)
{
  const Real & en2 = _par_vec[12];  //peak before first Sigma5
  const Real & en3 = _par_vec[13];  //first Sigma5
  const Real & en4 = _par_vec[14];  //peak between Sigma5's
  const Real & en5 = _par_vec[15];  //second Sigma5
  const Real & en6 = _par_vec[16];  //Sigma17

  const Real & th2 = _par_vec[17];  //position of peak before first Sigma5
  const Real & th4 = _par_vec[18];  //position of peak between Sigma5's
  Real th6 = 2 * std::acos(5.0 / std::sqrt(34.0));  //Sigma17 rotation angle

  Real a12 = 0.5;  //RSW shape factor. In previous versions, these were allowed to
  Real a23 = a12;  //vary, however there were too few vicinal boundaries in the
  Real a34 = a12;  //ensemble to constrain them. We found that forcing the great
  Real a45 = a12;  //majority of them to be 0.5 helped to constrain the fit and
  Real a56 = a12;  //produced reasonable results. This holds true for most of the
  Real a67 = a12;  //RSW shape factors throughout this code.

  Real en1 = 0;  //Sigma1 at left end
  Real en7 = 0;  //Sigma1 at right end

  Real th1 = 0;  //Sigma1 at left end
  Real th3 = std::acos(0.8);  //first Sigma5
  Real th5 = std::acos(0.6);  //second Sigma5
  Real th7 = libMesh::pi / 2;  //Sigma1 at right end

  //piecewise RSW function
  for (unsigned int i = 0; i < ksi.size(); i++)
  {
    if (ksi[i] <= th2)
      _en[i] = en1 + (en2 - en1) * rSW(ksi[i], th1, th2, a12);
    else if (ksi[i] > th2 && ksi[i] <= th3)
      _en[i] = en3 + (en2 - en3) * rSW(ksi[i], th3, th2, a23);
    else if (ksi[i] > th3 && ksi[i] <= th4)
      _en[i] = en3 + (en4 - en3) * rSW(ksi[i], th3, th4, a34);
    else if (ksi[i] > th4 && ksi[i] <= th5)
      _en[i] = en5 + (en4 - en5) * rSW(ksi[i], th5, th4, a45);
    else if (ksi[i] > th5 && ksi[i] <= th6)
      _en[i] = en6 + (en5 - en6) * rSW(ksi[i], th6, th5, a56);
    else if (ksi[i] > th6 && ksi[i] <= th7)
      _en[i] = en7 + (en6 - en7) * rSW(ksi[i], th7, th6, a67);
  }
}

void
AnisoGBEnergyUserObject::set110(std::vector<Real> & e110)
{
  const Real & pwr1 = _par_vec[19];  //110 tilt/twist mix power law: twist
  const Real & pwr2 = _par_vec[20];  //110 tilt/twist mix power law: tilt

  std::vector<Real> entwist(_geom110[0].size());
  std::vector<Real> entilt(_geom110[0].size());

  twist110(_geom110[1], entwist);
  aTGB110(_geom110[2], _geom110[1], entilt);

  for (unsigned int i = 0; i < _geom110[1].size(); ++i)
    e110[i] = entwist[i] * std::pow((1 - (2 * _geom110[3][i] / libMesh::pi)), pwr1),
    + entilt[i] * std::pow((2 * _geom110[3][i] / libMesh::pi), pwr2);
}

void
AnisoGBEnergyUserObject::twist110(std::vector<Real> & ksi, std::vector<Real> & entwist)
{
  const Real & th1 = _par_vec[21];  //110 twist peak position

  const Real & en1 = _par_vec[22];  //110 twist energy peak value
  const Real & en2 = _par_vec[23];  //Sigma3 energy (110 twist, so not a coherent twin)
  const Real & en3 = _par_vec[24];  //energy at the symmetry point

  Real a01 = 0.5;
  Real a12 = 0.5;
  Real a23 = 0.5;

  Real th2 = std::acos(1.0/3.0);  //Sigma3
  Real th3 = libMesh::pi/2;  //110 90-degree boundary is semi-special, although not a CSL
  Real period = libMesh::pi;  //the twist period

  for (unsigned int i = 0;i < ksi.size(); i++)
  {
    ksi[i] = std::fmod(std::abs(ksi[i]), period);  //rotation symmetry
    if (ksi[i] > period / 2)
      ksi[i] = period - ksi[i];
    if (ksi[i] <= th1)
      entwist[i] = en1 * rSW(ksi[i], 0, th1, a01);
    else if (ksi[i] > th1 && ksi[i] <= th2)
      entwist[i] = en2 + (en1 - en2) * rSW(ksi[i], th2, th1, a12);
    else if (ksi[i] > th2)
      entwist[i] = en3 + (en2 - en3) * rSW(ksi[i], th3, th2, a23);
  }
}

void
AnisoGBEnergyUserObject::aTGB110(std::vector<Real> & eta, std::vector<Real> & ksi,
std::vector<Real> & entilt)
{
  const Real & a  =  _par_vec[25];  //110 aTGB interpolation RSW shape factor

  Real period = libMesh::pi;

  std::vector<Real> en1(ksi.size());
  std::vector<Real> en2(ksi.size());

  std::vector<Real> temp(ksi.size());

  sTGB110(ksi, en1);
  for (unsigned int i = 0; i < ksi.size(); i++)
```

```
    temp[i] = period - ksi[i];
  sTGB110(temp, en2);

  //Power-law interpolation did not work well in this case. Did an RSW function instead
  for (unsigned int i = 0; i < ksi.size(); i++)
  {
    if (en1[i] >= en2[i])
      entilt[i] = en2[i] + (en1[i] - en2[i]) * rSW(eta[i], libMesh::pi, 0, a);
    else
      entilt[i] = en1[i] + (en2[i] - en1[i]) * rSW(eta[i], 0, libMesh::pi, a);
  }
}

void
AnisoGBEnergyUserObject::sTGB110(std::vector<Real> & ksi, std::vector<Real> & _en)
{
  const Real & en2 = _par_vec[26];  //peak between Sigma1 and Sigma3
  const Real & en3 = _par_vec[27];  //Coherent Sigma3 twin relative energy;
  //one of the more important element-dependent parameters
  const Real & en4 = _par_vec[28];  //energy peak between Sigma3 and Sigma11
  const Real & en5 = _par_vec[29];  //Sigma11 energy
  const Real & en6 = _par_vec[30];  //energy peak between Sigma11 and Sigma1

  const Real & th2 = _par_vec[31];  //peak between Sigma1 and Sigma3
  const Real & th4 = _par_vec[32];  //peak between Sigma3 and Sigma11
  const Real & th6 = _par_vec[33];  //peak between Sigma11 and higher Sigma1

  Real a12 = 0.5;
  Real a23 = a12;
  Real a34 = a12;
  Real a45 = a12;
  Real a56 = a12;
  Real a67 = a12;

  Real en1 = 0;
  Real en7 = 0;

  Real th1 = 0;
  Real th3 = std::acos(1.0 / 3.0);  //Sigma3
  Real th5 = std::acos(-7.0 / 11.0);  //Sigma11
  Real th7 = libMesh::pi;

  for (unsigned int i = 0; i < ksi.size(); i++)
  {
    ksi[i] = libMesh::pi - ksi[i];  //This is a legacy of an earlier (ksi,eta) mapping
    if (ksi[i] <= th2)
      _en[i] = en1 + (en2 - en1) * rSW(ksi[i], th1, th2, a12);
    else if (ksi[i] > th2 && ksi[i] <= th3)
      _en[i] = en3 + (en2 - en3)*rSW(ksi[i], th3, th2, a23);
    else if (ksi[i] > th3 && ksi[i] <= th4)
      _en[i] = en3 + (en4 - en3)*rSW(ksi[i], th3, th4, a34);
    else if (ksi[i] > th4 && ksi[i] <= th5)
      _en[i] = en5 + (en4 - en5) * rSW(ksi[i], th5, th4, a45);
    else if (ksi[i] > th5 && ksi[i] <= th6)
      _en[i] = en5 + (en6 - en5) * rSW(ksi[i], th5, th6, a56);
    else if (ksi[i] > th6 && ksi[i] <= th7)
      _en[i] = en7 + (en6 - en7) * rSW(ksi[i], th7, th6, a67);
  }

}

void
AnisoGBEnergyUserObject::set111(std::vector<Real> & e111)
{
  const Real & a = _par_vec[34];   //linear part of 111 tilt/twist interpolation
  Real b = a-1;   //ensures correct value at x = 1
  std::vector<Real> entwist(_geom111[1].size());
  std::vector<Real> entilt(_geom111[1].size());

  twist111(_geom111[1], entwist);
  aTGB111(_geom111[2], _geom111[1], entilt);

  //This one fit well enough with a simple one-parameter parabola that the more
  //complicated power laws in the other sets weren't needed
  for (unsigned int i = 0; i < _geom111[1].size(); ++i)
    e111[i]=entwist[i]+(entilt[i]-entwist[i]) * (a * 2 * _geom111[3][i]/libMesh::pi,
    - b * std::pow((2 * _geom111[3][i] / libMesh::pi), 2));
}

void
AnisoGBEnergyUserObject::twist111(std::vector<Real> & ksi, std::vector<Real> & entwist)
{
  const Real & thd = _par_vec[36];  //111 twist peak position

  const Real & enm = _par_vec[37];  //111 twist energy at the peak
  const Real & en2 = _par_vec[27];  //Coherent Sigma3 twin shows up in two distinct places
  //in the code
  const Real & a1 = _par_vec[35];  //111 twist RSW shape parameter
  const Real & a2 = a1;

  for (unsigned int i = 0; i < ksi.size(); ++i)
  {
    if (ksi[i] > libMesh::pi / 3)
      ksi[i] = 2 * libMesh::pi / 3 - ksi[i];
    if (ksi[i] <= thd)
      entwist[i] = enm * rSW(ksi[i], 0, thd, a1);
    else
      entwist[i] = en2 + (enm - en2) * rSW(ksi[i], libMesh::pi / 3, thd, a2);
  }
}

void
AnisoGBEnergyUserObject::aTGB111(std::vector<Real> & eta, std::vector<Real> & ksi,
std::vector<Real> & entilt)
{
  for (unsigned int i = 0; i < ksi.size(); ++i)
  {
    if (ksi[i] > libMesh::pi / 3)
      ksi[i] = 2 * libMesh::pi / 3 - ksi[i];
    if (eta[i] > libMesh::pi / 3)
      eta[i] = 2 * libMesh::pi / 3 - eta[i];
  }

  /* Below the following value of ksi, we ignore the eta dependence.  This is
  because there's little evidence that it actually varies.  Above this
  value, we interpolate on an RSW function that follows the Sigma3 line,
  which is also a line of symmetry for the function.
  */
  const Real & ksim = _par_vec[38];  //111 aTGB ksi break
  const Real & enmax = _par_vec[39];  //Energy at the peak (ksi == ksim)
  const Real & enmin = _par_vec[40];  //energy at the minimum (Sigma3, eta == 0)
  const Real & encnt = _par_vec[41];  //energy at the symmetry point (Sigma3, eta == pi/3)

  Real a1 = 0.5;
  Real a2 = 0.5;
  /* eta scaling parameter for 111 aTGB RSW function on Sigma3 line
  This RSW function is unusual in that the change in shape of the
  function is much better captured by changing the angular scale rather
  than changing the dimensionless shape factor.
  */
  const Real & etascale = _par_vec[42];
  Real chi;
  for (unsigned int i = 0; i < ksi.size(); ++i)
  {
    if (ksi[i] <= ksim)
      entilt[i] = enmax * rSW(ksi[i], 0, ksim, a1);
    else
    {
      //chi is the shape of the function along the Sigma3 line.
      chi = enmin + (encnt - enmin) * rSW(eta[i], 0, libMesh::pi / (2 * etascale), 0.5);
      entilt[i] = chi + (enmax - chi)*rSW(ksi[i], libMesh::pi / 3, ksim, a2);
    }
  }
}

Real
AnisoGBEnergyUserObject::rSW(Real theta, Real thetaMin, Real thetaMax, Real a)
{
  Real dtheta = thetaMax - thetaMin;  //Interval of angles where defined
  theta = (theta - thetaMin) * libMesh::pi / (dtheta * 2);  //Normalized angle

  //RSW function evaluation
  Real _en;
  if (std::sin(theta) >= 0.000001)  //Cut off a small sins to avoid 0*infinity problem.
  //The proper limit is 0.
    _en = std::sin(theta) - a * (std::sin(theta) * std::log(std::sin(theta)));
  else
    _en = std::sin(theta);
  return _en;
}
```

B.2 GBAnisoEnergy.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#include "GBAnisoEnergy.h"

template<>
InputParameters validParams<GBAnisoEnergy>()
{
  InputParameters params = validParams<Material>();
  params.addRequiredCoupledVar("T","Temperature in Kelvin");
  params.addParam<Real>("f0s", 0.125, "The GB energy constant ");
  params.addRequiredParam<Real>("wGB", "Diffuse GB width in nm ");
  params.addParam<Real>("length_scale", 1.0e-9, "Length scale in m, where
        default is nm");
  params.addParam<Real>("time_scale", 1.0e-9, "Time scale in s, where default is ns");
  params.addParam<Real>("GBMobility", -1, "GB mobility input in m^4/(J*s)");
  params.addParam<Real>("GBmob0", 0, "Grain boundary mobility prefactor in m^4/(J*s)");
  params.addParam<Real>("Q", 0, "Grain boundary migration activation energy in eV");
  params.addParam<Real>("molar_volume", 24.62e-6, "Molar volume in m^3/mol, needed
    for temperature gradient driving force");
  params.addRequiredCoupledVarWithAutoBuild("v", "var_name_base", "op_num", "Array
    of coupled variables");
  params.addRequiredParam<UserObjectName>("AnisoGBEnergy_UserObject","Where the map of
    the energies are held");
  return params;
}


GBAnisoEnergy::GBAnisoEnergy(const InputParameters & parameters) :
    Material(parameters),
    _f0s(getParam<Real>("f0s")),
    _wGB(getParam<Real>("wGB")),
    _length_scale(getParam<Real>("length_scale")),
    _time_scale(getParam<Real>("time_scale")),
    _GBmob0(getParam<Real>("GBmob0")),
    _Q(getParam<Real>("Q")),
    _GBMobility(getParam<Real>("GBMobility")),
    _molar_vol(getParam<Real>("molar_volume")),
    _nop(coupledComponents("v")),
    _T(coupledValue("T")),
    _sigma(declareProperty<Real>("sigma")),
    _M_GB(declareProperty<Real>("M_GB")),
    _kappa(declareProperty<Real>("kappa_op")),
    _gamma(declareProperty<Real>("gamma_asymm")),
    _L(declareProperty<Real>("L")),
    _l_GB(declareProperty<Real>("l_GB")),
    _mu(declareProperty<Real>("mu")),
    _entropy_diff(declareProperty<Real>("entropy_diff")),
    _molar_volume(declareProperty<Real>("molar_volume")),
    _act_wGB(declareProperty<Real>("act_wGB")),
    _tgrad_corr_mult(declareProperty<Real>("tgrad_corr_mult")),
    _aniso_GB_energy(getUserObject<AnisoGBEnergyUserObject>("AnisoGBEnergy_UserObject")),
    _kb(8.617343e-5) //Boltzmann constant in eV/K
{
  if (_GBMobility == -1 && _GBmob0 == 0)
    mooseError("Either a value for GBMobility or for GBmob0 and Q must be provided");

    _v.resize(_nop);
    for (unsigned int ops = 0 ; ops < _nop; ++ops)
    {
      _v[ops] = &coupledValue("v", ops);
    }
}

void
GBAnisoEnergy::computeQpProperties()
{
  Real M0 = _GBmob0;
  Real JtoeV = 6.24150974e18;// joule to eV conversion
  Real GBenergy = _aniso_GB_energy.getWeightedEnergy(_current_elem->id(), _qp);

//Convert to lengthscale^4/(eV*timescale);
  M0 *= _time_scale/(JtoeV*(_length_scale*_length_scale*_length_scale*_length_scale));
  _sigma[_qp] = GBenergy*JtoeV*(_length_scale*_length_scale);// eV/nm^2

  if (_GBMobility < 0)
    _M_GB[_qp] = M0*std::exp(-_Q/(_kb*_T[_qp]));
  else
    _M_GB[_qp] = _GBMobility*_time_scale/(JtoeV*(_length_scale*_length_scale*
        _length_scale*_length_scale)); //Convert to lengthscale^4/(eV*timescale)

  _l_GB[_qp] = _wGB; //in the length scale of the system
  _L[_qp] = 4.0/3.0*_M_GB[_qp]/_l_GB[_qp];
  _kappa[_qp] = 3.0/4.0*_sigma[_qp]*_l_GB[_qp];
  _gamma[_qp] = 1.5;
  _mu[_qp] = 3.0/4.0*1/_f0s*_sigma[_qp]/_l_GB[_qp];
  _entropy_diff[_qp] = 8.0e3*JtoeV; //J/(K mol) converted to eV(K mol)
  _molar_volume[_qp] = _molar_vol/(_length_scale*_length_scale*_length_scale);
  _act_wGB[_qp] = 0.5e-9/_length_scale;
  _tgrad_corr_mult[_qp] = _mu[_qp]*9.0/8.0;
}
```

B.3 ACGrGrAnis5DoF

```
#include "ACGrGrAnis5DoF.h"

template<>
InputParameters validParams<ACGrGrAnis5DoF>()
{
  InputParameters params = ACBulk<Real>::validParams();
  params.addClassDescription("Grain-Boundary model poly crystaline interface
        Allen-Cahn Kernel");
  params.addRequiredCoupledVarWithAutoBuild("v", "var_name_base", "op_num",
        "Array of coupled variables");
  params.addCoupledVar("T", "temperature");
  params.addParam<Real>("length_scale", 1.0e-9, "Length scale in m, where default is nm");
  params.addRequiredParam<unsigned int>("op","The order parameter number this is acting on");
  params.addRequiredParam<UserObjectName>("AnisoGBEnergy_UserObject","Where the map of
        the energies are held");
  params.addParam<Real>("f0s", 0.125, "The GB energy constant ");
  return params;
}

ACGrGrAnis5DoF::ACGrGrAnis5DoF(const InputParameters & parameters) :
    ACBulk<Real>(parameters),
    _aniso_GB_energy(getUserObject<AnisoGBEnergyUserObject>("AnisoGBEnergy_UserObject")),
    _mu(getMaterialProperty<Real>("mu")),
    _gamma(getMaterialProperty<Real>("gamma_asymm")),
    _tgrad_corr_mult(getMaterialProperty<Real>("tgrad_corr_mult")),
    _l_GB(getMaterialProperty<Real>("l_GB")),
    _has_T(isCoupled("T")),
    _grad_T(_has_T ? &coupledGradient("T") : NULL),
    _op(getParam<unsigned int>("op")),
    _ncrys(coupledComponents("v")),
    _length_scale(getParam<Real>("length_scale")),
    _f0s(getParam<Real>("f0s"))
{
  // Initialize values for crystals
  _vals.resize(_ncrys);
  _grad_vals.resize(_ncrys);


  for (unsigned int crys = 0; crys < _ncrys; ++crys)
  {
    // Initialize variables
    _vals[crys] = &coupledValue("v", crys);
    _grad_vals[crys] = &coupledGradient("v", crys);
  }
}

Real
ACGrGrAnis5DoF::computeDFDOP(PFFunctionType type)
{
  //start sums at zero
  Real SumEtaj = 0.0;
  Real SumEtaij = 0.0;
  Real SumEtaSigmaj = 0.0;
  Real SumEtaSigmaij = 0.0;
  Real Sum2Eta4Eta = 0.0;
  Real SumGradEta = 0.0;
  Real Dsigma_Deta;

  const std::map<std::pair<unsigned int, unsigned int>, Real> & gb_energy_map =
        _aniso_GB_energy.getGBEnergies(_current_elem->id());

  if (gb_energy_map.size() < 2) //only non-zero at multi-junctions
    Dsigma_Deta = 0.0;
  else
  {
    std::set<unsigned int> op_set;
    typedef std::map<std::pair<unsigned int, unsigned int>, Real>::const_iterator energies_it;
    for (energies_it it = gb_energy_map.begin(); it != gb_energy_map.end(); ++it)
    {
      unsigned int op1 = (it->first).first;
      unsigned int op2 = (it->first).second;
      op_set.insert(op1);
      op_set.insert(op2);
      Real sigmaij = it->second;
      SumEtaij += (*_vals[op1])[_qp] * (*_vals[op1])[_qp] * (*_vals[op2])[_qp] * (*_vals[op2])[_qp];
      SumEtaSigmaij += sigmaij * (*_vals[op1])[_qp] * (*_vals[op1])[_qp] *
            (*_vals[op2])[_qp] * (*_vals[op2])[_qp];
    }
    if (op_set.count(_op))
    {
      for (std::set<unsigned int>::const_iterator it = op_set.begin(); it != op_set.end(); ++it)
      {
        if (*it != _op)
        {
          Real sigmaiop = 0.0;
          SumEtaj += (*_vals[*it])[_qp] * (*_vals[*it])[_qp];
          if (_op > *it) // The lower order parameter is first in the pair
          sigmaiop = gb_energy_map.find(std::pair<unsigned int, unsigned int>(*it,_op))->second;
          else
          sigmaiop = gb_energy_map.find(std::pair<unsigned int, unsigned int>(_op,*it))->second;

          SumEtaSigmaj += sigmaiop * (*_vals[*it])[_qp] * (*_vals[*it])[_qp];
        }
        Sum2Eta4Eta += ((*_vals[*it])[_qp] * (*_vals[*it])[_qp] * (*_vals[*it])[_qp] *
                (*_vals[*it])[_qp])/4.0 - ((*_vals[*it])[_qp] * (*_vals[*it])[_qp])/2.0;
        SumGradEta += ((*_grad_vals[*it])[_qp] * (*_grad_vals[*it])[_qp]);
      }
      Dsigma_Deta = 2 * _u[_qp] * (SumEtaSigmaj * SumEtaij - SumEtaj * SumEtaSigmaij) /
            (SumEtaij * SumEtaij);
      Real JtoeV = 6.24150974e18;// joule to eV conversion
      Dsigma_Deta *= JtoeV * (_length_scale * _length_scale);// eV/nm^2
    }
    else
    Dsigma_Deta = 0.0;
  }

  Real tgrad_correction = 0.0;

  //Calcualte either the residual or jacobian of the grain growth free energy
  switch (type)
  {
    case Residual:
      if (_has_T)
        tgrad_correction = _tgrad_corr_mult[_qp]*_grad_u[_qp]*(*_grad_T)[_qp];
      return Dsigma_Deta * (6.0 / _l_GB[_qp] * (Sum2Eta4Eta + _gamma[_qp] * SumEtaij + 0.25)
            + 0.75 * _l_GB[_qp] * SumGradEta);
    case Jacobian:
      if (_has_T)
        tgrad_correction = _tgrad_corr_mult[_qp]*_grad_phi[_j][_qp]*(*_grad_T)[_qp];
      return _phi[_j][_qp]*Dsigma_Deta * (6.0 / _l_GB[_qp] * (Sum2Eta4Eta + _gamma[_qp] *
            SumEtaij + 0.25) + 0.75 * _l_GB[_qp] * SumGradEta);
  }

  mooseError("Invalid type passed in");
}

Real
ACGrGrAnis5DoF::computeQpOffDiagJacobian(unsigned int jvar)
{
  return 0.0;
}
```

Bibliography

[1] a. Michael R. Tonks, C. R. Stanek, D. Gaston, D. P. C. Millet, P. Nerikar, S. Du, a. A. David Anderson, and R. Williamson, “multiscale development of a fission gas thermal conductivity model: Coupling atomic, meso and continuum level simulations,” .

[2] M. R. Tonks, D. Gaston, P. C. Millet, D. Andrs, and P. Talbot, “An object-oriented finite element framework for multiphysics phase field simulations,” Computational Materials Science, 51(1):20-29 51 .

[3] G. S. Rohrer, “Grain boundary energy anisotropy: a review,” .

[4] N. Moelans, B. Blanpain, and P. Wollants, “Quantitative analysis of grain boundary properties in a generalized phase field model for grain growth in anisotropic systems,” .

[5] T. Harbison, “ANISOTROPIC GRAIN BOUNDARY ENERGY FUNCTION FOR URANIUM DIOXIDE,” .

[6] V. V. Bulatov, B. W. Reed, and M. Kumar, “Grain boundary energy function for fcc metals,” .

[7] D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandie, “MOOSE: A parallel

computational framework for coupled systems of nonlinear equations," Nuclear Engineering and Design, 239(10):17681778 (2009).

[8] G. Gottstein and L. Shvindlerman, "Triple junction drag and grain growth in 2D polycrystals," .