INTRODUCTION TO PHASE FIELD MODELING

by

Tyler S. Trogden

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 2019
Copyright © 2019 Tyler S. Trogden

All Rights Reserved
This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date

Evan Hansen, Advisor

Date

Kevin Kelley, Senior Thesis Coordinator

Date

Richard Hatt, Committee Member

Date

Lance Nelson, Committee Member

Date

Todd Lines, Chair
ABSTRACT

INTRODUCTION TO PHASE FIELD MODELING

Tyler S. Trogden
Department of Physics
Bachelor of Science

Phase-field modeling is introduced as a way to simulate grain growth evolution in systems of crystalline solids; this is achieved using MOOSE (Multi-Physics Object Oriented Simulation Environment) which was developed at Idaho National Laboratory (INL). It has been used to model grain growth of UO$_2$ at INL using an isotropic grain boundary energy model. This paper is a tutorial for students seeking to learn how to read and write input files with MOOSE syntax in order to test modifications that include anisotropic grain boundary energies.
ACKNOWLEDGMENTS

I would be remise if I did not thank all of my professors who sacrificed their time to help mold this paper into the work that it has become, and for blessing me with the opportunity to grow and stretch by challenging me to go further than I knew I could. I thank, Brother Evan Hansen for his tutelage over these years, especially as my thesis advisor and research mentor; for giving light to my understanding and for believing in my potential. Brother Richard Hatt as a member of my thesis committee for his contribution to making this work what it is and for his dedication to excellence in teaching over these years. Brother Lance Nelson for being a member of my thesis committee and giving his invaluable insight in how to effectively teach students the art of programming. And Brother Kevin Kelley for helping bring this paper to fruition as my undergraduate coordinator. I would also like to thank Dale and Ruth Hansen for their selfless gift so that I, a complete stranger, could further my experience and education in what I love. I thank the Office of Research Computing at BYU for access to the most powerful tool in our research arsenal, Marylou. I also thank Dr. Michael Tonks for his kind generosity in allowing me to study with him and his graduate student Leo Zhang at the University of Florida. And last but certainly not least I thank my dear wife Jenni, whom I love, for her unparalleled love and support; if there was anyone who worked harder than me over these few years to make all this possible, it was her, thank you.
# Contents

Table of Contents xi

List of Figures xiii

1 Introduction 1
   1.1 Background ................................................. 1
   1.2 Objective .................................................. 5

2 MOOSE Preliminaries 7
   2.1 Finite Element Method ....................................... 7
   2.2 Mesh ....................................................... 9
   2.3 Phase Field Model .......................................... 10
   2.4 MOOSE Syntax ............................................... 12
       2.4.1 MOOSE Blocks ......................................... 13

3 Two Circular Embedded Grains 19
   3.1 Physical Description ........................................ 19
   3.2 Phase Field Model .......................................... 20
       3.2.1 Mesh .................................................. 20
       3.2.2 GlobalParams and Modules .............................. 21
       3.2.3 ICs, Materials, and Preconditioning ................. 22
       3.2.4 Executioner and Outputs ............................... 23
       3.2.5 Running MOOSE ...................................... 24
   3.3 Results and Analysis ....................................... 27

4 Two Dimensional Polycrystal 31
   4.1 Physical Description ........................................ 31
   4.2 Phase Field Model .......................................... 32
       4.2.1 Mesh Adaptivity ..................................... 32
       4.2.2 Time Adaptivity .................................... 32
       4.2.3 Input File Gensis via Peacock ....................... 34
   4.3 Results and Analysis ....................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Three Dimensional Polycrystal</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Physical Description</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Results and Analysis</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>47</td>
</tr>
<tr>
<td>A</td>
<td>MOOSE Input Files</td>
<td>49</td>
</tr>
<tr>
<td>A.1</td>
<td>Two Circular Embedded Grains</td>
<td>49</td>
</tr>
<tr>
<td>A.2</td>
<td>Two Dimensional Polycrystal</td>
<td>52</td>
</tr>
<tr>
<td>A.3</td>
<td>Three Dimensional Polycrystal</td>
<td>58</td>
</tr>
<tr>
<td>B</td>
<td>The Calculus of Variations</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Figures

1.1 TRISO fuel kernel with exposed layers. ................................................. 2
1.2 Example of 2D phase-field model. ......................................................... 4
1.3 Order parameter values for diffuse grain boundary. ................................. 4

2.1 Visual of a finite element discretization. ................................................ 8
2.2 Various finite elements in two and three dimensions. ................................ 9
2.3 Example of MOOSE block syntax. .......................................................... 12
2.4 Example of MOOSE sub-block syntax. ..................................................... 12
2.5 Code from the phase field source file. .................................................... 14
2.6 Example of a Mesh block. .................................................................. 14
2.7 Example of a Executioner block. ............................................................ 15
2.8 Example of a Outputs block. ................................................................ 16
2.9 Example of a GlobalParams block. ......................................................... 16
2.10 Example of a Kernels block. ................................................................ 17
2.11 Example of a AuxKernels block. ............................................................ 17
2.12 Example of a BCs block. ................................................................ 18

3.1 Embedded circular grain. ................................................................. 20
3.2 The Peacock GUI in Input File view. .................................................... 25
3.3 The Peacock GUI in Execute view. ......................................................... 26
3.4 Example of a converging solver. ............................................................ 27
3.5 Variable animation options. ................................................................. 28
3.6 Color map options for animation. .......................................................... 28
3.7 Exodus Viewer tab at start up. ............................................................... 29
3.8 Evolved two-circular grain system without a “wireframe”. ..................... 29

4.1 Example of mesh adaptivity. ............................................................... 33
4.2 Peacock GUI without a specified input file. ........................................... 34
4.3 Mesh block initialization using Peacock. ............................................... 35
4.4 Example of changing block parameters using Peacock. ......................... 36
4.5 Example of modifying block parameters using the Peacock GUI. .......... 37
4.6 Block drop down menu containing available sub-blocks when using Pea-
cock. ................................................................................................. 37
4.7 Time-lapse of a 2D multi-grain system. ............................................... 40
5.1 3D polycrystal of five grains. ................................. 43
Chapter 1

Introduction

1.1 Background

Material science is the field of study devoted to understanding the properties of condensed matter and how it behaves under specific conditions. A material that is of particular interest is UO$_2$, it being a main component of nuclear fuels today due to its ease of fabrication and relative stability [1]. Fuel for nuclear reactors is made into small isotropic spheres (see Figure 1.1) [2] that are then placed into rods (cladding) that are arrayed in a reactor assembly. Between the fuel and cladding are small gaps that allow for thermal expansion and contraction [1]. A problem that occurs after prolonged use of the fuel rods is that they will begin to crack, thus exposing the fuel to the in-reactor moderator (often times H$_2$O) and the reactor must be shut down for refueling, a procedure that can cost millions of dollars each time. This occurs due to swelling of the fuel pebbles and the production of fissile byproducts, both of which cause the initial gap in the fuel rod assembly to close and thus creating a bi-axial tensile stress [1] that can damage the fuel cladding.

Supercomputer simulations of the evolution of nuclear fuel has helped save millions
of dollars by being able to predict when fuel longevity has been maximized and before failure has been reached. A widely used method for modeling this evolution is called phase-field modeling as seen in Figure 1.2. A phase-field model is one that models grains in a solid by treating them as if they each were their own phase. We typically refer to a phase as in a liquid, solid or gas, but in this context the different phases are due to the lattice structures of each of the grains, or lattice impurities (i.e. dopants or vacancies); therefore, each grain and/or impurity is its own phase. In order to keep track of these grains, or order parameters ($\eta_i$), they are assigned a value of either a zero or a one over a spatial grid. A value of zero for $\eta_k$ corresponds to being “outside” of that given order parameter, and a value of one being “inside”. In most models however, the change between being either in or out of a given order parameter is not abrupt but gradual. Therefore, a transition point corresponds to a continuum of values from one to zero which is referred to as a diffuse boundary between order parameters [3] (see Figure 1.3). Because of the mis-orientation of these grains, relative to one another, the grain boundary (GB) exists, and associated with that boundary
1.1 Background

an energy.

This energy arises due to the relative positions of the atoms of neighboring grains and naturally will have a lowest energy configuration, which may be reached given that there is an activation energy large enough to allow the GBs to migrate. While GB energy is an integral part of the total energy of a sample of fuel, it is not in fact the entire contribution. Other contributions to the energy of the sample arise from internal characteristics such as energy stored in chemical and nuclear bonds. This total energy is known as the Gibbs free energy and is by definition the energy gained if the system were to be annihilated, or the energy required if suddenly it were “poofed” into existence [4]. Minimization of this energy is what governs the motion of the GBs and the partial differential equations that govern this process are the Cahn-Hilliard and Allen-Cahn equations, which will be introduced shortly in chapter 2.

The research that is being done at BYU-Idaho under the Materials Research Group (MRG) has the ultimate goal of being able to model grain growth evolution due to an anisotropic GB energy (i.e. energy that is dependent on the orientation of GBs relative to one another). The data required to do this model fitting is being gathered using molecular dynamics (MD) software (on supercomputers at the Office of Research Computing at BYU) and has the ultimate goal of finding an analytical function for the anisotropic GB energy. In order to model grain growth evolution a phase-field model must be utilized using software known as MOOSE, a multi-physics object oriented simulation environment, that was developed at Idaho National Laboratory (INL). Implementation of an anisotropic GB energy in MOOSE will need to be added. Once an analytic function is determined for the anisotropic GB energy, it can be implemented into MOOSE in order to run our simulations and give more accurate results regarding how the evolution of the system will occur.
Chapter 1 Introduction

Figure 1.2 Example of a 2D phase-field model where each individual grain is represented by $\eta_i$ and each grain boundary by the solid lines: https://www.mdpi.com/2075-4701/1/1/16/htm.

Figure 1.3 Illustration from Moelans et al. of values for order parameters considering a diffuse grain boundary with varying energies.
1.2 Objective

In order to incorporate the aforementioned changes and be able to interpret results of such an incorporated anisotropic GB model, this paper has been produced to introduce the reader to MOOSE syntax, conventions, and features, including the MOOSE GUI, Peacock. In order to begin and get the most benefit from this work the reader will need the following: a working knowledge of Linux syntax and a current version of MOOSE installed on their machine. For a short, but helpful, Linux tutorial, refer to the following: https://marylou.byu.edu/documentation/unix-tutorial/.

A current version of MOOSE may be downloaded manually from GitHub, but it is encouraged that the user use the step-by-step download from MOOSE Framework at INL: https://www.mooseframework.org/getting_started/index.html. Also the reader will want a text editor such as emacs or vim to edit input files, but it is encouraged that the latest version of atom be obtained since it will be referred to and used during the tutorial sections of this paper. A link to the current version of atom may be found here: https://atom.io/. Once atom is successfully downloaded it will also be necessary to add the appropriate MOOSE packages (every available package that says MOOSE) in order to use a very useful feature, auto completion.

Each example is expected to be a tutorial for the reader to work through as he or she reads, so that the code may be executed in real time and the physical system analyzed. Doing so the reader will soon find that a major part of the MOOSE syntax is the use of blocks and sub-blocks, whose syntax will be introduced in 2.4. Three examples have been prepared and naturally increase in complexity as the reader progresses, they are as follows:

1) two circular embedded grains,

2) two dimensional polycrystal, and
3) three dimensional polycrystal.

The first example will be given as an exercise in basic MOOSE syntax and in order for the reader to familiarize him- or herself with various block parameters. The last two examples will be given as an introduction to the MOOSE GUI Peacock, where the reader may become acquainted with a simpler way to create input files. Also, a brief physical description of each system will be given in the chapter that it is introduced. However, all of the examples follow basic physical principles that will be described in chapter 2.
Chapter 2

MOOSE Preliminaries

2.1 Finite Element Method

The entire framework of MOOSE is based on a method for discretizing very large systems into very many infinitesimally small ones. This is known as the Finite Element Method (FEM), as the name suggests. Imagine trying to tackle a problem as large and complex as knowing how each grain of UO$_2$ inside of nuclear fuel rods is influenced by in-pile (i.e. in-reactor) conditions; obviously this problem is so complex that perhaps one may think that it simply isn’t done, however with the FEM it goes from being impossible to doable.

For clarity on how the FEM is implemented consider a large bridge that is subject to the weight of traffic. The center of the bridge that has the least support and the largest load is subject to the greatest stress, and as one nears the edges of the bridge, where there is more support, the stress dissipates. Modeling how this bridge reacts under a load is complex in that we simply can’t have a single model for it, considering that each part is distinct in its reaction. However, if we simply take very small pieces of the bridge to model and then couple together those individual pieces using their
underlying equations of motion and stress, then soon we’ll have a model for the whole (see Figure 2.1).

The phase-field model is implemented in a similar way in MOOSE with its corresponding equations being for grain evolution which are then used to model whole systems of grains on large scales. Really the only limit to the uses of MOOSE is computational memory and time; systems that are simply too large to handle (i.e. the number of discretized pieces is on the order of tens of millions or more) may take too long to solve, even on modern HPC (High Performance Computing) clusters. Part of this complexity has to do with the discretized pieces themselves. In using the FEM we must pick a shape for the pieces we wish to model and then piece them all together with their respective and coupled equations solved; these discretized pieces are known as mesh and is one of the geniuses of the FEM.
2.2 Mesh

Mesh is the commonly used name for the discretized pieces used in FEA (Finite Element Analysis) collectively, but mesh also has subparts known as nodes and elements. Elements are the discretized pieces themselves which are usually a specific (and uniform) shape. Returning to the example of the bridge, it may suit us to break the bridge up into many tiny rectangles or triangles (considering just a two dimensional example) and then calculate the stress on each individual piece seeing how it affects its neighbor. Considering the boundary, the elements that are located there must satisfy the boundary conditions for a given problem, and are necessary to solve the problem as a whole. The accuracy of the solution for a given iteration also corresponds to the number of elements in a mesh; with increasing number the mesh will become more fine and thus better defined (see 4.2.1).

Nodes occur where each element is coupled to its neighbor, and the number of nodes per element may be different depending on the system and the accuracy of solution that is desired (see Figure 2.2 ). It is here at these points where the governing PDEs are solved and can be used iteratively to find a solution to the problem as a whole.

![Finite Elements](https://www.comsol.com/multiphysics/finite-element-method)

**Figure 2.2** Several examples of finite elements both in two and three dimensions, where nodes are visible as dots: https://www.comsol.com/multiphysics/finite-element-method.
Chapter 2 MOOSE Preliminaries

2.3 Phase Field Model

The equation that governs the evolution of the order parameters, implemented in MOOSE, is known as the Allen-Cahn equation:

$$\frac{\partial \eta_i}{\partial t} = -L_i \frac{\delta F}{\delta \eta_i}, \quad i = 1, 2, 3...N$$ (2.1)

where $L_i$ is the associated GB mobility, the free-energy functional is represented by $F$ [5], and the presence of $\delta$ indicates a variational derivative [6].

Mobility may be best described with the following, simplified, equation:

$$L_i = \frac{1}{m_i}$$ (2.2)

where $m_i$ is the mass of our object of interest. While 2.2 is not exactly the model that is implemented in MOOSE, it does serve as a good illustration as to what GB mobility means. As seen in 2.2, if $m_i$ is small, then $L_i$ is large and vice-versa, meaning that as $m_i \to 0$ the force required to move the GB decreases and thus the migration of the GB will be very fast. In other words, the larger the mobility the easier it is to “push” and “pull” the GB, the opposite being true for $m_i \to \infty$. The GB mobility may also be a function of grain mis-orientation and inclination, i.e. $L_i = L_i (\bar{\theta}, \bar{\phi})$, where $\bar{\theta}$ is a three component vector that gives the relative mis-orientation between adjacent grains, and $\bar{\phi}$ is a two (or three) component vector that gives their relative inclination [3].

The free-energy functional, seen in 2.1, as proposed by Moelans et al. and which is implemented in MOOSE is of the form:
2.3 Phase Field Model

\[ F = \int_V f(\eta_1, \ldots, \eta_p, \nabla \eta_1, \ldots, \nabla \eta_p) \, dV \]
\[ = \int_V \left[ m f_0(\eta_1, \eta_2, \ldots, \eta_p) + \frac{\kappa}{2} \sum_{i=1}^{p} (\nabla \eta_i)^2 \right] \, dV \]  

(2.3)

\( \kappa \) being the energy gradient coefficient and where \( \nabla \eta_i \) is the gradient of the \( i \)-th order parameter, which points in the direction of the greatest change in \( \eta_i \).

The function \( f_0 \) being proposed as:

\[ f_0(\eta_1, \eta_2, \ldots, \eta_p) = \sum_{i=1}^{p} \left( \frac{\eta_i^4}{4} - \frac{\eta_i^2}{2} \right) + \gamma \sum_{i=1}^{p} \sum_{j>i}^{p} \eta_i^2 \eta_j^2 + \frac{1}{4} \]  

(2.4)

where \( \gamma \) is a dimensionless constant that may vary also, depending on our analysis of the GB properties being either isotropic or anisotropic; the factor of 1/4 is necessary to maintain a diffuse GB. For further discussion on the physical description of the free-energy functional see Moelans et al [3].

The kinetics of a system of grains are such that the free-energy is minimized; this is the reason for the appearance of the variational derivative ¹ in 2.1. This minimization of energy manifests itself in the “growing” and “shrinking” of grains, which decreases the GB surface of some and increases it for others, whichever will minimize the total free-energy of the system. Individual grains are not annihilated, however, since the law of conservation of mass still plays a significant role here. Instead grains that “disappear” are assimilated by neighboring ones; i.e., their lattice structure, or phase, is aligned to match that of the parent grain, to create a perfect crystal which has a lower energy state. This GB migration is a consequence of the applied force due to a potential energy ( \( \vec{F}_{orce} = -\nabla V \) ) associated with the relative position between grains, and can be calculated using various potential energy models but the one that has been found to be closest to observed phenomena is the Basak potential [7].

¹See Appendix B.
2.4 MOOSE Syntax

In order for a simulation to run correctly and represent physical phenomena, several blocks and sub-blocks are necessary and will be briefly introduced in this section. Although there are various blocks and sub-blocks (with many needed for a given problem) only six fundamental blocks will be mentioned here. The syntax for all MOOSE blocks, however, follow basic universal rules. As an example refer to figure 2.3.

```
1  [Mesh]
2      type = GeneratedMesh
3      dim = 2
4      xmax = 2
5      ymax = 2
6      nx = 20
7      ny = 20
8          uniform_refine = 2
9  ]
```

**Figure 2.3** Example of MOOSE block syntax using as the example a mesh block.

```
[./Modules]
[./PhaseField]
[./GrainGrowth]
    mobility = L
    kappa = kappa_op
[./]/
[./]/
[
```

**Figure 2.4** Example of a MOOSE sub-block nested inside of the modules block. Notice also how there are multiple sub-blocks one nested inside of the other.

Notice that each block begins and ends with hard brackets (i.e. [ ] ), where the first brackets contain the block name and the last are left empty. MOOSE sub-blocks are exactly the same with the only exception that within the first hard brackets, the sub-block name is led by a ./ and the closing hard brackets are not empty but contain
.\. to close the sub-block. Also, remember that sub-blocks are only initialized inside of a block.

After each block/sub-block initialization, indentation is required showing that the body of that block, or sub-block, pertains to it. The body is used to specify the desired parameter values for our simulation. Note that variable names are hard coded and should not be changed when writing an input file, since a change in any variable name will result in the simulation failing to start. This is a consequence of the hard code that is being used to run the simulation. The code is written in C and can be seen in the source code folder inside of the phase field module in MOOSE. To find what variables to use simply look up a similar problem from an other input file or check the MOOSE documentation online https://mooseframework.org/syntax/.

A handy way to find where these variables are being used in the C code, is to simply hit Cmd-Shift-F (or the analogous Windows command) in an atom window with any specified phase field directory’s input file open. From there atom will not only tell you where in the input file the variable is but also where it is used in the entire project.

2.4.1 MOOSE Blocks

Mesh

Typically the first block that is declared in an input file is the Mesh block. As the name implies, it is used to determine what type of mesh will be used to solve and visualize the problem that we’ll be handling. It is also where the user defines the dimension and grid size for the problem. MOOSE is so versatile that you can have custom mesh generated from your own files, this is in fact the default type when creating your problem. However MOOSE offers a library, libMesh, where one can choose from an already created mesh type.
Chapter 2 MOOSE Preliminaries

```cpp
#include "AllenCahn.h"

registerMooseObject("PhaseFieldApp", AllenCahn);

template <>
InputParameters validParams<AllenCahn>()
{
    InputParameters params = ACBulk<Real>::validParams();
    params.addClassDescription("Allen-Cahn Kernel that uses a DerivativeMaterial Free Energy");
    params.addRequiredParam<MaterialPropertyName>
        ("f_name", "Base name of the free energy function F defined in a DerivativeParsedMaterial");
    return params;
}

AllenCahn::AllenCahn(const InputParameters &parameters)
:   ACBulk<Real>(parameters),
    _nvar(_coupled_moose_vars.size()),
    _dFdEta(getMaterialPropertyDerivative<Real>("f_name", _var.name())),
    _d2FdEtdarg(_nvar)
{
    // Iterate over all coupled variables
    for (unsigned int i = 0; i < _nvar; ++i)
        _d2FdEtdarg[i] = GetMaterialPropertyDerivative<Real>("f_name", _var.name(), _coupled_moose_vars[i]->name());
}
```

**Figure 2.5** Example of the C code in the phase field source file. This particular file is the kernel that evaluates the Allen-Cahn equation.

```
1 | [Mesh]
2 | type = GeneratedMesh
3 | dim = 2
4 | xmax = 2
5 | ymax = 2
6 | nx = 20
7 | ny = 20
8 | uniform_refine = 2
9 | []
```

**Figure 2.6** Example of a Mesh block initialization for two dimensions.

Executioner

The **Executioner**, typically called towards the end of an input file, is where the user defines what type of solver they would like to use. MOOSE comes with several kinds of solvers, the two types of solvers that are available are for steady-state problems or
time dependent ones know as transient state. The tolerance for the linear and non-linear iterations are chosen, these determine how close is close enough when deciding whether our solution in that time step has converged or not, and also the maximum time for which the problem will be considered is chosen.

![Figure 2.7 Example of a Executioner block initialization.](image)

**Outputs**

The **Outputs** block is primarily responsible for, you guessed it, the type of output that MOOSE will give. There are several files that can be outputted but the most common being `.csv` and ExodusII files. Options for **Outputs** also allows the user to decide when MOOSE should create an output file, e.g. when a time step is finished or every third time step, etc. Perhaps the most useful files for our purposes will be the `.e` files (the file extension for ExodusII files), which we will use as input in order to create videos using the Peacock GUI.

**GlobalParams**

As the reader may see, this block is simply where we define the number of order parameters (`op_num`). Here the user may specify however many grains that are necessary
for their system, and it may be wise to also consider the dimensions of the sample that the grains will occupy, if there will be many grains (i.e. upwards of twenty five) then perhaps consider using a different model.

Kernels and AuxKernels

Here is where the user defines what equations MOOSE will automatically look for to implement, however, it is assumed that the user is familiar with the problem they care to solve so that they may call for the correct kernels and auxiliary kernels. For example, to solve a phase field problem we would have to use:

\[
R_{\eta} = \left( \frac{\partial \eta_j}{\partial t}, \psi_m \right) + (\nabla(\kappa \eta_j), \nabla(L \psi_m)) + L \left( \frac{\partial f_{loc}}{\partial \eta_j} + \frac{\partial E_d}{\partial \eta_j}, \psi_m \right) 
\]  

(2.5)

which is the residual for the Allen-Cahn equation and each term in parentheses represents a different kernel. In this case the kernels are: TimeDerivative, ACInterface, and AllenCahn, respectively, where each kernel is calculating a volume integral of an inner product, with notation similar to that of Dirac notation. So as the reader can
see MOOSE has a lot going on in the background that developers have painstakingly made available so that we simply get to run the problems we care about.

![Figure 2.10 Example of a Kernels block initialization.](image)

![Figure 2.11 Example of a AuxKernels block initialization.](image)

**BCs**

In order to simplify our model we typically choose boundary conditions that are periodic so that we may avoid the complexities that are occurring at the edges of our sample. As the reader will see, a system with periodic boundary conditions will cause grains to wrap around the sample when coming to an interface where the sample is discontinuous; this feature may be modified to specify in which direction the user would like these interfaces to wrap.

As the reader will note from introductory courses in differential equations for boundary value problems, there are numerous types of boundary conditions that we
may want to apply to our system, an important example being either Dirichlet or Neumann. These boundary conditions specify how the function is behaving on the boundary or how it is changing on the boundary, respectively, where these conditions may be homogenous or not. An extensive list of conditions may be found at https://www.mooseframework.org/syntax/BCs/index.html#bcs-system where each condition is categorized into a specific module for MOOSE.

![Figure 2.12 Example of a BCs block initialization.](image)
Chapter 3

Two Circular Embedded Grains

3.1 Physical Description

The system that we will be analyzing in this chapter is that of a circular grain embedded inside of another circular grain. This problem is one that is of some importance in the MRG because this is an analogous system to the one we use in order to calculate GB energies using LAAMPS. It allows us to find the energy given any mis-orientation and inclination of the GBs relative to one another. The migration of atoms from one boundary to another, in this system, is due to a grain’s inner and outer radii. In a circular grain there is both an inner and outer boundary, given that the boundary is diffuse, and migration occurs from the convex side to the concave side, said to occur due to the chemical potential being higher on one side than on the other [8]. Therefore, as mentioned before one grain will grow at the expense of the other, typically the smaller grains being the ones that shrink.
3.2 Phase Field Model

While creating your own input file you will be walked through how to initialize several blocks not previously mentioned as well as those that were, the reason for this is because every block is not used in each input file, and so the most basic were introduced first and specific ones will be introduced as needed. When you have finished the input file you may use appendix A.1 to check your work, it is encouraged that you look at the appendix only after trying to write the whole input file yourself. To begin, go ahead and open up a new file in atom and give it the extension .i; the first block that we will initialize is the Mesh block.

3.2.1 Mesh

In typical MOOSE syntax start the block and then add the following variables (you may play with whatever values you’d like for the parameters): dim (for the dimension
of our problem), \(x_{\text{max}}, y_{\text{max}}, n_x\) (number of discretized steps in the x-direction), \(n_y\) (number of discretized steps in the y-direction), and \texttt{uniform_refine}\ (the initial mesh refinement). There is another special piece of information that MOOSE will need to tell it where to get the mesh, the keyword is \texttt{type} set equal to the mesh library we want to use, which for our purposes will always be \texttt{GeneratedMesh}.

### 3.2.2 GlobalParams and Modules

Next we’ll need a \texttt{GlobalParams} block to specify the number of order parameters that will be used. The variable names are: \texttt{op_num} (for the number of order parameters) and \texttt{var_name_base} (the display name for each grain with its respective number).

We will also go ahead and initialize the \texttt{Modules} block, this simply lets MOOSE know what equations we’ll be using to solve this system, it is analogous to \texttt{Kernels} and \texttt{AuxKernels} and may be used in their place for this example. Here we will also need two sub-blocks, one telling the module that we’re using and the other what type of problem we’re solving in that module. Remember that sub-blocks are only called within a block, and the type of problem we’re solving will be a sub-block of the first sub-block (if this is confusing or your simulation does not seem to run correctly, go ahead and take a look at the \texttt{Modules} block in appendix A.1). The only variables that it will need are the parameters to the sub-sub-block, and which specify constants for the Allen-Cahn and the free-energy equations, they are: \texttt{mobility} (set equal to \(L\)) and \texttt{kappa} (set equal to \texttt{kappa_op}). The names of the sub-blocks are \texttt{PhaseField} and \texttt{GrainGrowth}, in that order. Remember to close each sub-block appropriately and finally close the block as well.
3.2.3 ICs, Materials, and Preconditioning

The ICs block is where we will specify the initial conditions of our system. Our system at hand is simply a circular grain inside of a much larger one as seen in 3.1, which has its own special sub-block in MOOSE due to its popularity as an introductory problem. This sub-block will be nested inside of the sub-block PolyCrystalICs, it being the wider category of polycrystalline problems. The only things that are left for us to specify are the radius of our inner grain, its interfacial width, and where we want to start it initially, those specific variables are: radius, int_width, x, and y, respectively. This also may be a good time to mention something you may have already noticed if you downloaded atom along with the corresponding MOOSE packages, and that is that while you type in atom the variables names will show up for use in auto-completion by hitting the tab key, once you’ve given the first letter or so of the block, sub-block, or variable name.

The Materials block is usually where we will specify what material we are using along with its corresponding properties, such as: temperature of the material, its GB mobility, activation energy, GB energy (being isotropic in these cases we take it to be only a constant for now), and the GB width. In our case we will only be concerned here with the GBs so we can use a helpful sub-block called gb with the special type GBEvolution and the following parameters: T, Gmob0, Q (the activation energy), GBenergy, and wGB. We will also need to specify the scales we will be working in, namely what length and time. The scales for a phase field system are considered macroscopic; therefore, we will want to set length_scale to one micron ($10^{-6}$) and time_scale to one second.

The pre-conditioner is an optional block that is used to pre-condition the matrix that will be used to solve our system of PDEs (this comes from the Jacobian of the JFNK method, for more on this see [9]). The Preconditioning block takes one
sub-block, SMP, with two options, full and the type, which are set to true and SMP, respectively.

### 3.2.4 Executioner and Outputs

Next we will initialize the Executioner block and add two special sub-blocks Adaptivity and TimeStepper, which control mesh and time adaptivity, respectively. We start by telling MOOSE what type of problem we’re solving, transient or steady-state, and how to solve it. Then we’ll upload the necessary tools to do so from the PETSc library and specify how long to run with given tolerances. First, the type of problem we are solving is, naturally, Transient since we are watching the evolution of the grains in time, and we’ll use as the solve_type the NEWTON method (for more information please see https://mooseframework.org/).

As mentioned above we will be using tools from the PETSc library in order to solve our problem, the necessary PETSc options can be found in A.1 lines fifty-nine and sixty. The parameters for how we want the problem to be solved however, may be specified, they are: \texttt{nl\_rel\_tol, nl\_max\_its, l\_tol, and l\_max\_its}. These parameters tell MOOSE how small we desire the error to be for both linear and non-linear solves along with how many iterations to perform before deciding that a solution is divergent. And lastly we specify how long we desire the solver to run with \texttt{end\_time}.

To initialize the mesh adaptivity sub-block we need to specify: how many adaptive steps we want to take given the initial conditions, how many times a single mesh element may be refined, and how to coarsen the mesh or make the mesh finer. The parameters are as follows: \texttt{initial\_adaptivity, max\_h\_level, refine\_fraction, and coarsen\_fraction}. You may choose values for these as you like but note that you will want to refine and coarsen the mesh ever so slightly on a given adaptive step.
Chapter 3 Two Circular Embedded Grains

The time adaptivity block will require (depending on the type of time-step adaptivity) that we give it: an initial time-step, the number of optimal non-linear iterations to maintain if the time-step is adapted, and the factors to increase or decrease the time-step by. The parameter names for each of these are as follows: \textit{dt}, \textit{optimal_iterations}, \textit{growth_factor}, and \textit{cutback_factor}. As mentioned we will also need to specify the type of adaptive time-step, which for our case we will want to be based on the number of iterations at each step; our best option then for this will be \texttt{IterationAdaptiveDT}.

Last but not least we will tell MOOSE how we would like the data from the simulation to be packaged for output. Their are several options and I’d encourage the reader to test out a few to see what they do (file types and descriptions can be found here \url{https://mooseframework.org/syntax/Outputs/index.html}). The file types that we will use for our simulations are called \texttt{ExodusII} files (files with a \texttt{.e} extension) which are used to animate our simulation in realtime. To tell MOOSE these are the files we are looking for we simply initialize the block and assign \texttt{true} to the \texttt{exodus} parameter.

3.2.5 Running MOOSE

With the previous blocks instantiated the reader can now send the file as input into the Peacock GUI. This can be achieved by typing the following command into the Linux terminal \texttt{peacock -i yourInputFile.i}. This particular command will only work if the Peacock executable is in the file path however, otherwise the reader will need to find the directory containing the executable by ascending the file path and entering a new branch (i.e. \texttt{./././python/peacock/peacock -i yourInputFile.i} for example). To make Peacock accessible while in any of the directories add the following: \texttt{export PATH=/Users/username/Downloads/projects/moose/python/peacock:$PATH}
3.2 Phase Field Model

to your `.bash_profile`. In order for this change to take affect it may be necessary to restart Linux; now try executing again.

When this last step is performed successfully a window like the one seen in figure 3.2 should appear. Amazingly, from this single GUI one can edit the system being tested just as if they were modifying the input file.

![Peacock GUI](image)

Figure 3.2 Peacock GUI where on the left hand side, at any time before initialization of the simulation, the system parameters may be changed by simply double clicking the desired block.

In order to start the simulation the reader will simply need to navigate to the `Execute` tab of the GUI, which should appear the same as in figure 3.3. While in this tab the reader will notice a column of boxes; the first tells you the directory from which your input file came, and the second tells the location of your `peacock` executable.

Two below that there is a handy box for allocating the number of processors that the user would like to run the calculations on, so it may be a good idea to know
the number of processors on your laptop. Once the user has determined the number they would like to allocate then they simply need to hit \textit{Run} and the executioner will begin.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure33.png}
\caption{Peacock GUI in the \textit{Execute} tab where the number of cores to be used may be specified at the \textit{Use MPI} line.}
\end{figure}

In figure 3.3 the reader will note that the solver is iterating over linear and non-linear steps, as previously mentioned. Dark green indicates that the solver is converging on a more and more well defined value; this will keep happening either until the number of iterations has reached the predetermined maximum or the tolerance is met, i.e. the solution converged as seen in figure 3.4. Once this occurs another iterative process will start for a new time step and adaptive options (if they were specified); this will repeat until the determined time, after which (or during) the \textit{Exodus Viewer} tab will have our final results in the form of an animation of the system being analyzed.
3.3 Results and Analysis

While in the Exodus Viewer tab there are many optional features that the user may specify in order to better visualize the system. One can choose between which variables to display as in figure 3.5 and which color map to use as in figure 3.6. Also the mesh “wireframe” may be removed, using the “Mesh” options box, in order to better visualize the grains and grain boundaries.

In the bottom center of the GUI there is a “play” button which allows the user to step forward and backward in time in order to watch the simulation evolve. This can be activated even before the solver has finished executing so that the user may watch the evolution as the solver converges. As expected the smaller grain is “shrinking”, decreasing the total grain boundary energy and causing the boundary to disappear.

With the particular variables being “bnds” (GBs) the color bar to the right of the animation is trying to show that as you approach the center of the cell you transition from one grain to the other. At the boundary is where that transition occurs, hence
Chapter 3 Two Circular Embedded Grains

Figure 3.5 From the pull down tab in the variables window the user may display a choice of variable; here \texttt{bnds} stands for the grain boundaries and \texttt{gr} for the respective grains.

Figure 3.6 The user may also specify the color mapping for the chosen variable from a wide variety, even \texttt{grayscale}.

if you are not either fully in nor out of your respective grain the value is a continuum.
Figure 3.7 In the Exodus Viewer tab the user can visualize the evolution of the system in real time. In this case there are two grains, one enveloped by the other, and where their boundary can be seen in black.

Figure 3.8 Closeup of an evolved two-grain system without a "wireframe".
Chapter 4

Two Dimensional Polycrystal

4.1 Physical Description

In this chapter the reader will be introduced to a way to streamline the creation of input files by utilizing features of Peacock. Also mesh and time adaptivity will be explained as yet another way to help speed up analysis of a grain growth simulation. The input file that will be referenced is a default file, that comes with each MOOSE installation, titled `grain_growth_2D_voronoi_newadapt.i`, as seen in appendix A.2. Here we will generalize our results from chapter 3 to multi-grain systems in two dimensions. Our system will consist of a rectangular cross section with multiple grains that are packed tightly together (i.e. there are no voids in the crystal). This type of system is created using a Voronoi tessellation (a repeating pattern that has no gaps between figures) and is a pseudo-random process that creates GBs depending on the initial points that are generated using a random seed. The conditions under which this system will be subject are standard and the kinetics of the system are identical to the ones mentioned in chapter 3.
4.2 Phase Field Model

4.2.1 Mesh Adaptivity

Mesh adaptivity is a powerful option that MOOSE incorporates into a system that allows the executioner to run more efficiently by decreasing the number of linear and non-linear solves per mesh element. This can be seen visually in Peacock as the number of mesh elements either decreases in an area where the solution is well defined and increases where it is not, hence putting computational resources where they are most needed.

As the reader may see in 4.1 the successive time step coarsened the grain mesh inside of the order parameters where the solution to each was well defined and also refined the mesh that was closer to the boundaries where the solutions are constantly changing. This makes for a faster execution when solving and is a great tool to conserve time and computing power.

To implement adaptive mesh one must simply add an Adaptivity block to their input file (or sub-block inside of the Executioner block), with the proper sub-blocks identifying what to change and how to change it. Typically when using the phase field module for modeling grain growth the variables that we want to apply adaptivity to are the grain boundaries.

4.2.2 Time Adaptivity

Time adaptivity is akin to mesh adaptivity in that it puts resources where they are needed during computation in order to run more efficiently. The time step (\(dt\)) is changed, for each mesh element, depending on convergence of a solution during a given time step and is changed at the beginning of a new time step, either to be shorter or longer. Shorter time steps correspond to solutions that need greater accuracy, and
4.2 Phase Field Model

Figure 4.1 The top and bottom images are of the same multi-grain system whose snapshot was taken at the same time step. The top image displays the mesh wireframe in order to illustrate the point that when using the optional mesh adaptivity the parts of the system that require more refinement also have a finer mesh while the parts of the system that are well defined use a coarser mesh.

the opposite it true for a longer adapted time step. To implement time adaptivity as an option in MOOSE one must call the sub-bock inside of the Executioner.
4.2.3 Input File Gensis via Peacock

To begin, first navigate to the `grain_growth` directory just below the `examples` directory under the phase field module in MOOSE. From here open up Peacock without an input file by simply typing `peacock` and execute. This will cause the Peacock GUI to open up as seen in figure 4.2:

![Figure 4.2 Peacock GUI without a specified input file.](image)

Notice that every possible MOOSE block appears just to the right of the figure window and may be selected using the small checkbox next to its name; we will be using these boxes to create our input files. There is one catch however, though we may be able to specify whichever MOOSE block we like for use in our input file, not all of the sub-blocks appear as options to choose from, and so we will need to create a saved input file from the Peacock, which will be demonstrated below.

Begin by first selecting each major block that we will be using in our input file, these blocks are (in no particular order): `Mesh`, `GlobalParams`, `UserObjects`, `ICs`, `Variables`, `AuxVariables`, `Kernels`, `AuxKernels`, `BCs`, `Materials`, `Postprocessors`,
4.2 Phase Field Model

Executioner, Adaptivity, and Outputs. For some of these blocks we will be able to specify their parameters along with their sub-blocks completely, for others however we will need to edit them using atom as we did in chapter 3, as mentioned.

To give the parameters for a certain block simply double-click on the block you’d like to edit, for example start with the Mesh block; once you’ve double-clicked the name Mesh on the right side of the Peacock GUI the following window should appear:

![Figure 4.3 Mesh block initialization using Peacock.](image)

Apart from avoiding having to type in every block name with all the the opening and closing hard brackets, using Peacock to create an input file also saves us from having to know all of the variable names. As the reader can see in 4.3 each of the parameters for a given block appear as specifiable options, in order to change them all that has to be done is to double-click that option and type in the desired value or option for that parameter. When it comes to specifying the type for a given block, most blocks will have a drop down menu at the very top of the window with options to choose from, as seen in 4.4 (recall that we will only be using *type=GeneratedMesh*
for our purposes).

![Block parameter window](image)

**Figure 4.4** Block parameters may also be changed using the Peacock GUI by simply double-clicking on the desired block, and then modifying the values by using the pop-up window or the drop down menu tabs.

From here we may also specify the other given parameters that are pertinent to our problem (for the parameters that are not used in our problem simply do not pick them and they will not show up in your input file). For example, click the drop down menu next to `dim` to choose the correct dimension for the problem (see figure 4.5), also choose `REPLICATED` for the parallel_type which is necessary to specify BCs for this problem. You may continue in this way specifying values for each of the same parameters that were specified for the mesh block back in chapter 3, and once you are done simply hit **Apply & Close** at the bottom-left hand side of the window.

Next the reader should also specify any available sub-blocks by using the drop down menus that appear next to each of the block names in the Peacock GUI start-up window (for example the Adaptivity block as seen in figure 4.6).

The following sub-blocks will need to be specified and will be left for the reader
Figure 4.5 Drop down tabs, where available, may be used to change block parameter values, and where no tab is available one may double click the value cell adjacent to the variable name.

Figure 4.6 Any available sub-blocks for a given block are located under that block’s drop down menu, as seen here for the Adaptness block.

to find inside of each respective block, in order to familiarize him- or herself with the the drop down menus. Also note that if the sub-block is not in any of the drop down menus (which a few will not be) then don’t worry, these are the sub-blocks that will have to be specified manually using atom as before. The sub-blocks that will be needed are: Indicators, Markers, Periodic, TimeStepper, PolycrystalICs, PolycrystalKernel, PolycrystalVariables, voronoi, bnds, bnds_aux, top_bottom, CuGrGr, dt, and bound_adapt and errorfrac.

Once each of the sub-blocks that were available from the drop down menus have been found go ahead and give the value and options for each of the block’s parameters using Peacock as shown above (for variable names see appendix A.2). Once that has been done save your input file by clicking save as from the Input File drop down menu at the top-left of the Peacock GUI start-up window (you will want to save it in
a directory that is below where your phase field executable directory is found). Now from here we can add any missing sub-blocks along with their parameters.

As the reader should have noticed the following sub-blocks were not available from any of the drop down menus mentioned above: **voronoi** (inside the **UserObjects** block), **bnds** (inside the **AuxVariables** block), **bnds_aux** (inside the **AuxKernels** block), **top_bottom** (inside the **BCs** block), **CuGrGr** (inside the **Materials** block), **dt** (inside the **Postprocessors** block), **error** (inside the **Indicators** sub-block), **bound_adapt** and **errorfrac** (inside the **Markers** sub-block). These are the ones that will needed to be specified using **atom**. So go ahead and open up the file that was previously saved and let’s start editing.

At first the reader will notice that the syntax is a not exactly correct, i.e. the sub-blocks are not declared exactly right and will need to be fixed. This can be done all at once using the **Find & Replace** feature that is available in **atom**. To open **Find & Replace** simply hit **Cmd-F**; now to change each sub-block name so that it appears with the correct ./ beginning, type “(space) [” into the **Find** bar replaced by “(space) [./”. By hitting **Replace All** every instance of whatever was placed in the **Find** bar will be changed. Now to correct the sub-block closing brackets do the same thing as before except inside of the **Find** bar put “(space) [./]” and replace it with “(space) [../]”.

Now that the existing code has been corrected lets add the omitted sub-blocks as mentioned above into their respective blocks (or sub-blocks in case of nested sub-blocks). Again you may reference appendix A.2 for the appropriate variable names and specifications. Once those have been added we may now use the same Linux command from above to upload the file into Peacock and let it run.
4.3 Results and Analysis

Now let us start the simulation in the typical fashion as was shown in the previous chapter by typing the following command into Terminal from the folder holding your input file: `peacock -i yourInput File.i`. But before we do this I suggest modifying the input file’s run time to some larger time interval, in order to better see the grain evolution. Once the Peacock GUI pops up go ahead and begin execution from the `Execute` tab by selecting the number of processors you wish to use and then hitting `Run`.

Once the maximum time for execution is reached go ahead and enter the `Exodus Viewer` tab and play or step through the time steps to see the grain growth animation in real time. You’ll notice, as we’ve seen before, that the smaller grains are definitely shrinking while the larger grains are growing, essentially “taking over” the smaller grains. Once you have successfully followed to this point you should have a figure similar to the one shown in 4.7.
Figure 4.7 Completed grain growth evolution model starting from the initial sample, to the fully evolved grain. Notice how the triangular grain in the bottom of the top picture when fully evolved has been assimilated by the time the simulation is complete.
Chapter 5

Three Dimensional Polycrystal

5.1 Physical Description

In the previous chapters we analyzed systems in two dimensions for two-grain and multi-grain systems. Here we will generalize results to three dimensions with multiple grains. As a reference the reader may use the default MOOSE file `grain_growth_3D.i` found in appendix A.3. Once again the system will consist of a polycrystal at standard conditions with a sample of five embedded grains (much more than five grains in three dimensions can take a very long time to run on an ordinary laptop). It is expected that, as mentioned in chapter 3, each grain will evolve so as to minimize the total energy of the entire sample by decreasing the total area of grain boundary per grain.

This simulation, like the last, is also created using a Voronoi tessellation algorithm, only generalized to three dimensions; therefore, the code to run this simulation is nearly identical to the code from the last chapter, except for the addition of a few sub-blocks inside of the `AuxVariables` and `AuxKernels` blocks. Those additions are mainly for being able to keep track of more GBs and order parameters, and so creating the input file, using Peacock, will be left as an exercise for the reader.
Chapter 5 Three Dimensional Polycrystal

5.2 Results and Analysis

In order to run simulations in Peacock for a three dimensional system it is advised that the reader lower the number of grains. Alternatively one may submit this simulation as a batch job to a HPC in order to run the simulation in a reasonable amount of time. For members of the MRG to submit batch jobs to the Office of Research Computing one must simply create a SLURM script and specify their desired resources along with the code to run the job, the script generator can be found at the Office of Research Computing home page.

Once the simulation is complete we may analyze the system using Peacock. If you used a HPC to run the job then it will be necessary for you to make a secure copy of your output files back to your personal computer in order to use Peacock. The following command will allow you to do so using Linux: `scp username@from_host:file.e /local/directory/`, where the `-r` flag may be added as an option in order to recursively copy the entire directory if desired.

Once this step is complete we can now go ahead and visually analyze our sample and see its evolution in time. As we would have expected the sample minimized the total grain boundary surface area by incorporating smaller grains into larger ones by causing a phase change, as seen in the time lapse images of figure 5.1.
Figure 5.1 Three dimensional polycrystal in rectangular mesh.
Chapter 6

Conclusion

Summary

During the course of this tutorial we set out to help the reader be able to understand MOOSE syntax and implement it in a phase filed model scenario. The reader was introduced to such syntax as is seen in typical MOOSE input files that follows a “block” format with the most essential blocks taking emphasis, such as: Mesh, Variables, Kernels, BCs, Executioner, AuxKernels, BCs, GlobalParams, and Outputs, with several adaptive options such as mesh and time adaptivity. The reader was also introduced to the MOOSE GUI Peacock along with some of its features, which allows for visualization of grain evolution in time and space, and streamlining of input file genesis. All of this was done by analyzing simple systems of grains from two to multi-grain structures in two and three dimensions.
Chapter 6 Conclusion

Future Work

Considerable work is being done in the field of material science and with MOOSE as a powerful tool it is foreseeable that the field will continue to grow and give new insight and solutions to problems encountered in engineering and physics. One area of interest, as previously mentioned, that could make a particularly large impact on research and application, is the development of the anisotropic GB energy model. Currently an anisotropic GB energy model is being used in a MOOSE application named MARMOT, which can only be used by authorized individuals and is protected software. Developing an analytic model of the grain boundary energy as a function of inclination and mis-orientation is a future endeavor of the Material Research Team and once developed and proven will need to be implemented into MOOSE through creation of a new MOOSE kernel and possible new application, that can be open-sourced.
Bibliography


Brigham Young University - Idaho, 2016.
Appendix A

MOOSE Input Files

A.1 Two Circular Embedded Grains

```plaintext
[Mesh]
  type = GeneratedMesh
  dim = 2
  xmax = 2
  ymax = 2
  nx = 20
  ny = 20
  uniform_refine = 2

[GlobalParams]
  op_num = 2
  var_name_base = gr

[Modules]
```
Chapter A MOOSE Input Files

[PhaseField]
[./GrainGrowth]
  mobility = L
  kappa = kappa_op
[../]
[../]
ICs
[./PolycrystalICs]
[./BicrystalCircleGrainIC]
  radius = 0.3
  x = 1.0
  y = 1.0
  int_width = 0.03
[../]
[../]
Materials
[./gb]
  type = GBEvolution
  T = 1700
  GBmob0 = 9.2124e-09
  Q = 2.77
  GBenergy = 1.58
  length_scale = 1e-6
  time_scale = 1.0
  wGB = 0.025
[../]
[../]

A.1 Two Circular Embedded Grains

[Preconditioning]
[./SMP]
  type = SMP
  full = true
[../]

[Executioner]
  type = Transient
  solve_type = NEWTDN
  petsc_options_iname = '-pc_type -pc_hypre_type -ksp_gmres_restart -
                         pc_hypre_boomeramg_strong_threshold'
  petsc_options_value = 'hypre boomeramg 31 0.7'
  nl_rel_tol = 1e-08
  nl_max_its = 10
  l_tol = 1e-4
  l_max_its = 30
  end_time = 500

[./Adaptivity]
  initial_adaptivity = 3
  max_h_level = 4
  refine_fraction = 0.9
  coarsen_fraction = 0.05
[../]

[./TimeStepper]
  type = IterationAdaptiveDT
  dt = 1 # Initial time step. In this simulation it changes.
optimal_iterations = 4 # Time step will adapt to maintain this number of nonlinear iterations
linear_iteration_ratio = 8
iteration_window = 0
growth_factor = 1.2
cutback_factor = 0.75

[Outputs]
exodus = true
print_perf_log = true

A.2 Two Dimensional Polycrystal

Input files in A.2 and A.3 come with each installation of MOOSE and can be found in the appropriate file path: moose/modules/phase_field/examples/grain_growth.
dim = 2 # Problem dimension
nx = 12 # Number of elements in the x-direction
ny = 12 # Number of elements in the y-direction
nz = 0 # Number of elements in the z-direction
xmin = 0 # minimum x-coordinate of the mesh
xmax = 1000 # maximum x-coordinate of the mesh
ymin = 0 # minimum y-coordinate of the mesh
ymax = 1000 # maximum y-coordinate of the mesh
zmin = 0
zmax = 0
elem_type = QUAD4 # Type of elements used in the mesh
uniform_refine = 3 # Initial uniform refinement of the mesh
parallel_type = replicated # Periodic BCs

[GlobalParams]

# Parameters used by several kernels that are defined globally to simplify input
    file
op_num = 15 # Number of grains
var_name_base = gr # Base name of grains

[UserObjects]
.[/voronoi]
    type = PolycrystalVoronoi
    grain_num = 15
    rand_seed = 42
    coloring_algorithm = bt # Forces the UserObject to assign one grain to each op
.[../]
Chapter A MOOSE Input Files

41
42 [ICs]
43 [./PolycrystalICs]
44 [./PolycrystalColoringIC]
45 polycrystal_ic_uo = voronoi
46 [...] [...]
48 []
49
50 [Variables]
51 # Variable block, where all variables in the simulation are declared
52 [./PolycrystalVariables]
53 # Custom action that created all of the grain variables and sets their initial condition
54 [...] [...]
55 []
56
57 [AuxVariables]
58 # Dependent variables
59 [./bnds]
60 # Variable used to visualize the grain boundaries in the simulation
61 [...] [...]
62 []
63
64 [Kernels]
65 # Kernel block, where the kernels defining the residual equations are set up.
66 [./PolycrystalKernel]
67 # Custom action creating all necessary kernels for grain growth. All input parameters are up in GlobalParams
68 [...] [...]
69 []
A.2 Two Dimensional Polycrystal

[AuxKernels]

# AuxKernel block, defining the equations used to calculate the auxvars
[./bnds_aux]

# AuxKernel that calculates the GB term
type = BndsCalcAux
variable = bnds
execute_on = timestep_end
[../]
[

[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 = GBEvolution # Quantitative material properties for copper grain growth.

Dimensions are nm and ns
GBmob0 = 2.5e-6 # Mobility prefactor for Cu from Schonfelder1997
GBenergy = 0.708 # GB energy for Cu from Schonfelder1997
Q = 0.23 # Activation energy for grain growth from Schonfelder 1997
T = 450 # Constant temperature of the simulation (for mobility calculation)
wGB = 14 # Width of the diffuse GB
[../]
[Postprocessors]

# Scalar postprocessors
[./dt]

# Outputs the current time step
type = TimestepSize
[../]

[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'

petsc_options_iname = '-pc_type -pc_hypre_type -ksp_gmres_restart -mat_mffd_type ,
petsc_options_value = 'hypre boomeramg 101 ds'

l_max_its = 30 # Max number of linear iterations
l_tol = 1e-4 # Relative tolerance for linear solves
nl_max_its = 40 # Max number of nonlinear iterations
nl_abs_tol = 1e-11 # Relative tolerance for nonlinear solves
nl_rel_tol = 1e-10 # Absolute tolerance for nonlinear solves

[./TimeStepper]
type = SolutionTimeAdaptiveDT
A.2 Two Dimensional Polycrystal

```plaintext
dt = 25  # Initial time step. In this simulation it changes.

start_time = 0.0
end_time = 4000
num_steps = 15

[Adaptivity]
marker = errorfrac
max_h_level = 4

[./Indicators]
[./error]
    type = GradientJumpIndicator
    variable = bnds

[./Markers]
[./bound_adapt]
    type = ValueThresholdMarker
    third_state = DO NOTHING
    coarsen = 1.0
    refine = 0.99
    variable = bnds
    invert = true

[./errorfrac]
    type = ErrorFractionMarker
    coarsen = 0.1
    refine = 0.7
```
A.3 Three Dimensional Polycrystal

# This simulation predicts GB migration of a 2D copper polycrystal with 100 grains represented with 18 order parameters
# Mesh adaptivity and time step adaptivity are used
# An AuxVariable is used to calculate the grain boundary locations
# Postprocessors are used to record time step and the number of grains

<Mesh>
  # Mesh block. Meshes can be read in or automatically generated
  type = GeneratedMesh
  dim = 3 # Problem dimension
  nx = 10 # Number of elements in the x-direction
  ny = 10 # Number of elements in the y-direction
  nz = 10
  xmin = 0 # minimum x-coordinate of the mesh
A.3 Three Dimensional Polycrystal

14  xmax = 1000  # maximum x-coordinate of the mesh
15  ymin = 0    # minimum y-coordinate of the mesh
16  ymax = 1000 # maximum y-coordinate of the mesh
17  zmin = 0
18  zmax = 1000
19  uniform_refine = 2  # Initial uniform refinement of the mesh
20
21  parallel_type = replicated # Periodic BCs
22
23
24 [GlobalParams]
25  # Parameters used by several kernels that are defined globally to simplify input
26  # file
27  op_num = 20  # Number of order parameters used
28  var_name_base = gr # Base name of grains
29
30 [Variables]
31  # Variable block, where all variables in the simulation are declared
32  [.//PolycrystalVariables]
33  [../]
34
35 [UserObjects]
36  [.//voronoi]
37  type = PolycrystalVoronoi
38  grain_num = 20  # Number of grains
39  rand_seed = 10
40  coloring_algorithm = bt
41  [../]
42  [./grain_tracker]
Chapter A  MOOSE Input Files

44 type = GrainTracker
45 threshold = 0.2
46 connecting_threshold = 0.08
47 compute_halo_maps = true  # Only necessary for displaying HALOS
48 [...]
49 []
50
51 [ICs]
52 [.PolycrystalICs]
53 [.PolycrystalColoringIC]
54   polycrystal_ic_uo = voronoi
55 [...]
56 [...]
57 []
58
59 [AuxVariables]
60  # Dependent variables
61 [.bnds]
62   # Variable used to visualize the grain boundaries in the simulation
63 [...]
64 [.unique_grains]
65   order = CONSTANT
66   family = MONOMIAL
67 [...]
68 [.var_indices]
69   order = CONSTANT
70   family = MONOMIAL
71 [...]
72 [.ghost_regions]
73   order = CONSTANT
74   family = MONOMIAL
A.3 Three Dimensional Polycrystal

```
[../]
[./halos]
  order = CONSTANT
  family = MONOMIAL
[../]
[]

[Kernels]
  # Kernel block, where the kernels defining the residual equations are set up.
  [./PolycrystalKernel]
  # Custom action creating all necessary kernels for grain growth.
  # All input parameters are up in GlobalParams
  [../]
  []

[AuxKernels]
  # AuxKernel block, defining the equations used to calculate the auxvars
  [./bnds_aux]
  # AuxKernel that calculates the GB term
  type = BndsCalcAux
  variable = bnds
  execute_on = 'initial timestep_end'
  [../]
  [./unique_grains]
  type = FeatureFloodCountAux
  variable = unique_grains
  flood_counter = grain_tracker
  field_display = UNIQUE_REGION
  execute_on = 'initial timestep_end'
  [../]
  [./var_indices]
```
Chapter A   MOOSE Input Files

106  type = FeatureFloodCountAux
107  variable = var_indices
108  flood_counter = grain_tracker
109  field_display = VARIABLE_COLORING
110  execute_on = 'initial timestep_end'
111  [../]
112  [./ghosted_entities]
113  type = FeatureFloodCountAux
114  variable = ghost_regions
115  flood_counter = grain_tracker
116  field_display = GHOSTED_ENTITIES
117  execute_on = 'initial timestep_end'
118  [../]
119  [./halos]
120  type = FeatureFloodCountAux
121  variable = halos
122  flood_counter = grain_tracker
123  field_display = HALOS
124  execute_on = 'initial timestep_end'
125  [../]
126  []
127
128  [BCs]
129  # Boundary Condition block
130  [./Periodic]
131  [./top_bottom]
132  auto_direction = 'x y'  # Makes problem periodic in the x and y directions
133  [../]
134  [../]
135  []
136
A.3 Three Dimensional Polycrystal

```
[Materials]
[./CuGrGr]
# Material properties
type = GBEvolution
T = 450 # Constant temperature of the simulation (for mobility calculation)
wGB = 40 # Width of the diffuse GB
GBmob0 = 2.5e-6 #m^4(Js) for copper from Schoenfelder1997
Q = 0.23 #eV for copper from Schoenfelder1997
GBenergy = 0.708 #J/m^2 from Schoenfelder1997

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

[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 101 ds'

l_max_its = 30 # Max number of linear iterations
l_tol = 1e-4 # Relative tolerance for linear solves
nl_max_its = 40 # Max number of nonlinear iterations
nl_rel_tol = 1e-10 # Absolute tolerance for nonlinear solves

start_time = 0.0
dt = 25 # Initial time step. In this simulation it changes.
optimal_iterations = 6 # Time step will adapt to maintain this number of nonlinear iterations

[./TimeStepper]
  type = IterationAdaptiveDT

[../]

[./Adaptivity]

# Block that turns on mesh adaptivity. Note that mesh will never coarsen beyond initial mesh (before uniform refinement)
initial_adaptivity = 2 # Number of times mesh is adapted to initial condition
refine_fraction = 0.6 # Fraction of high error that will be refined
coarsen_fraction = 0.1 # Fraction of low error that will coarsened
max_h_level = 3 # Max number of refinements used, starting from initial mesh (before uniform refinement)

[../]
A.3  Three Dimensional Polycrystal

192  exodus = true  # Exodus file will be outputted
193  csv = true
194  [./console]
195  type = Console
196  max_rows = 20  # Will print the 20 most recent postprocessor values to the
197     screen
198  [../]
Appendix B

The Calculus of Variations

Variational derivative is a term that comes from a branch of mathematics known as the calculus of variations. The technique is a familiar one that comes from an introductory course in calculus; I refer to finding the extrema of a function by taking derivatives of its independent variables and setting the derivative equal to zero. The same is true when trying to find the extrema for a functional, but now instead of maximizing or minimizing with respect to an independent variable you are optimizing with respect to a function.

Often times the mapping of a function \( f \) onto \( \mathbb{R} \) by another function \( I \) (i.e. \( f : I[f] \mapsto \mathbb{R} \)) represents an integral equation, e.g.:

\[
 f \mapsto I[f] = \int_V H(f(\bar{r}), f'(\bar{r}), \ldots, \bar{r}) \, dV \quad (B.1)
\]

where \( \bar{r} \) may represent a set of independent variables, each of which may be optimized. If there exists some small variation in our function, say \( \epsilon \) such that \( f^\epsilon \) is of the form:
\[ \tilde{f}(\bar{r}, \epsilon) = f(\bar{r}) + \epsilon \cdot \phi(\bar{r}) \]  
\[ \tilde{f}'(\bar{r}, \epsilon) = f'(\bar{r}) + \epsilon \cdot \phi'(\bar{r}) \]  

then we may optimize B.1 with respect to \( \epsilon \) as follows:

\[ \frac{\partial I}{\partial \epsilon} \bigg|_{\epsilon=0} = \frac{\delta I}{\delta \epsilon} \bigg|_{\epsilon=0} = \int_V \frac{\partial}{\partial \epsilon} \left[ H(\tilde{f}(\bar{r}, \epsilon), \tilde{f}'(\bar{r}, \epsilon), \ldots, \bar{r}) \right] dV = 0 \]  

As a consequence of the chain rule we have:

\[ \frac{\partial H}{\partial \epsilon} = \frac{\partial H}{\partial \tilde{f}} \frac{\partial \tilde{f}}{\partial \epsilon} + \frac{\partial H}{\partial \tilde{f}'} \frac{\partial \tilde{f}'}{\partial \epsilon} + \ldots \]
\[ = \frac{\partial H}{\partial \tilde{f}} \phi(\bar{r}) + \frac{\partial H}{\partial \tilde{f}'} \phi'(\bar{r}) + \ldots \]  

Therefore substituting B.5 into B.4 and applying integration by parts on a multi-variable function we get:

\[ \frac{\delta I}{\delta \epsilon} \bigg|_{\epsilon=0} = \int_V \left[ \frac{\partial H}{\partial \tilde{f}} - \frac{\partial}{\partial \tilde{r}} \left( \frac{\partial H}{\partial \tilde{f}'} \right) \right] \phi(\bar{r}) dV = 0 \]  

where \( \bar{r} \) is shorthand for a set of derivatives of independent variables. Noting, that since the integral B.6 is equal to zero, this implies that the integrand must be zero as well, leading to the well known Euler-Lagrange equations:

\[ \frac{\partial H}{\partial \tilde{f}} - \frac{\partial}{\partial \tilde{r}} \left( \frac{\partial H}{\partial \tilde{f}'} \right) = 0 \]