Percolation Behavior of Diffusionally Evolved Two-Phase Systems Simulated Using Phase Field Methods

by

Victor Eric Brunini

Submitted to the Department of Materials Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2008

© Victor Eric Brunini, MMVIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 18 2008
LIBRARIES

Author .....
Department of Materials Science and Engineering
May 9, 2008

Certified by .....
W. Craig Carter
Professor of Materials Science and Engineering, MacVicar Faculty Fellow
Thesis Supervisor

Certified by .....
Christopher A. Schuh
Danae and Vasilios Salapatas Associate Professor of Metallurgy
Thesis Supervisor

Accepted by .....
Caroline A. Ross
Chair, Undergraduate Committee

Percolation Behavior of Diffusionally Evolved Two-Phase Systems Simulated Using Phase Field Methods

by

Victor Eric Brunini

Submitted to the Department of Materials Science and Engineering
on May 9, 2008, in partial fulfillment of the
requirements for the degree of
Bachelor of Science

Abstract

Percolation is an important phenomenon that dramatically affects the properties of many multi-phase materials. As such, significant prior work has been done to investigate the percolation threshold and critical scaling exponents of randomly assembled composites. However many materials are non-random as a result of correlations that are introduced during processing. This work seeks to address this case by studying the percolation behavior of diffusionally evolved two phase systems. Specifically, the values of the percolation threshold and critical exponents ν, β, and γ are presented for two dimensional systems evolved through spinodal decomposition and nucleation and growth.

Thesis Supervisor: W. Craig Carter
Title: Professor of Materials Science and Engineering, MacVicar Faculty Fellow

Thesis Supervisor: Christopher A. Schuh
Title: Danae and Vasilios Salapatas Associate Professor of Metallurgy

Acknowledgments

I would like to acknowledge my advisors, Professors Carter and Schuh, for starting me on this project, giving me the freedom to tackle it on my own, and being available to help whenever I asked. I would like to thank Ming Tang for his help with numerical methods for solving partial differential equations, and Dan Cogswell for sharing some of his phase field simulation code for me to modify for this project.

Contents

Chapter 1

Introduction

Materials design is an important and effective method of surmounting the technological challenges of the 21st century. Microstructural engineering of multi-phase systems is becoming an increasingly common method of developing new materials to meet specific needs. In order for a microstructural engineer to effectively design materials it is essential to understand how changes in material composition affect microstructure, and in turn material properties.

This makes understanding the percolation transition in multi-phase systems very important for microstructural engineering because properties often transition rapidly around the percolation threshold. For example, consider the case of a two phase system where one phase is highly conductive and the other is significantly less so. As the phase fraction of the conductive phase increases across the percolation threshold the conductivity of the sample will increase dramatically with a small change in the phase fraction of the conductive phase [7, 10]. Many other properties in multi-phase systems exhibit similar sharp transitions around the percolation threshold.

A great deal of work has been done to study the percolation behavior of a variety of both continuum and lattice systems. Both site and bond percolation on a two dimensional square lattice, the Bethe lattice, and several lattices of higher dimension are well understood [11]. In addition continuum percolation of randomly placed discs with both a single radius and a distribution of radii, along with the three dimensional extension to spheres, is well studied [2, 6, 4, 13, 8]. While these cases provide useful

analogs for the microstructures of some materials, and a great deal has been learned from them about percolation behavior in general, they are all essentially random. That is, they do not account for the effects of physical phase correlations that are the result of microstructural evolution.

The goal of this work is to study the percolation behavior of diffusionally evolved systems in order to address the effects of the resulting physical phase correlations. Specifically two dimensional, two phase systems created through spinodal decomposition and nucleation and growth are discussed. While some simplifying assumptions are made, for example that the free energies are isotropic, the use of microstructural evolution simulations instead of simple random placement should more accurately reflect the behavior of many multi-phase materials.

Chapter 2

Theoretical Background

2.1 Phase Field Modeling

Phase field modeling involves the use of time evolution equations for conserved and non-conserved order parameters in combination with a free energy function to produce simulations displaying behavior often observed in microstructural evolution [9]. As the nature of the specific free energy function used will depend on the behavior that one is interested in simulating, the time evolution equations for conserved and non-conserved order parameters are the heart of the phase field method.

The evolution of a conserved order parameter, c, is governed by the Cahn-Hilliard equation:

$$\frac{\partial c}{\partial t} = M_c \left[\nabla^2 \frac{\partial F}{\partial c} - \epsilon_c^2 \nabla^4 c \right] \tag{2.1}$$

where M_c is a positive kinetic coefficient, F is the free energy function, and ϵ_c is a parameter determining the energy penalty due to concentration gradients [1]. Non-conserved order parameter evolution is described by the Allen-Cahn equation:

$$\frac{\partial \psi}{\partial t} = -M_\psi \left[\frac{\partial F}{\partial \psi} - \epsilon_\psi^2 \nabla^2 \psi \right] \tag{2.2}$$

where ψ is the order parameter, M_ψ is a positive kinetic coefficient, F is the free energy function, and ϵ_ψ is a parameter affecting interface width [14].

In this work the phase field method is used to simulate two types of microstructural evolution, spinodal decomposition and nucleation and growth. The specific free energy models used for each case are detailed in the following sections.

2.1.1 Spinodal Decomposition

In order to simulate spinodal decomposition a single conserved order parameter, c_b, representing the concentration of one component in a two component system with a dual well free energy function may be used. One such free energy function is:[9]

$$F(c_b) = \frac{16F_{max}}{(c_\beta - c_\alpha)^4}[(c_b - c_\alpha)(c_b - c_\beta)]^2 \tag{2.3}$$

This free energy function, plotted in Figure 2-1, has stable concentrations at c_α and c_β, and spinodal decomposition will be observed in the region where $\frac{\partial^2 F}{\partial c_b^2} < 0$. Differentiating twice gives:

$$\frac{\partial^2 F}{\partial c_b^2} = \frac{16F_{max}}{(c_\beta - c_\alpha)^4}[12c_b^2 - 12c_b(c_\alpha + c_\beta) + 2(c_\alpha^2 + 4c_\alpha c_\beta + c_\beta^2)]$$

Setting $\frac{\partial^2 F}{\partial c_b^2} = 0$ and solving for c_b yields:

$$c_b = \frac{c_\alpha + c_\beta}{2} \pm \frac{\sqrt{3(c_\alpha^2 - 2c_\alpha c_\beta + c_\beta^2)}}{6}$$

Therefore using $c_\alpha = 0.119$ and $c_\beta = 0.881$ spinodal decomposition will occur for $0.28 < c_b < 0.72$. This range is sufficiently wide to use this free energy function to determine the percolation threshold for spinodal decomposition.

2.1.2 Nucleation and Growth

Simulating nucleation and growth is more complex than spinodal decomposition since any simple dual well free energy function will exhibit spinodal decomposition for some range of compositions. In order to allow the simulation of nucleation and growth for any initial composition a second non-conserved order parameter, ψ, is used. ψ rep-

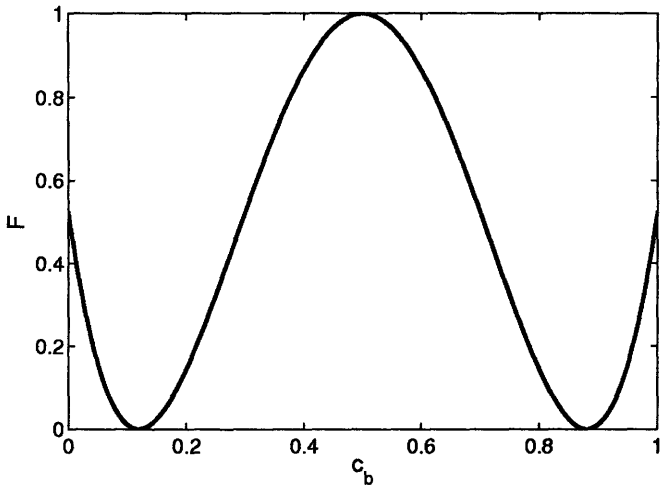


Figure 2-1: Dual well free energy function from Equation (2.3), with stable concentrations $c_\alpha = 0.119$ and $c_\beta = 0.881$

resents the phase of the system at a given point. Additionally, a new free energy function, dependent on both c_b and ψ is needed. One such model for a binary eutectic system is proposed by Wheeler, McFadden and Boettinger [1]. The simulations performed in this work are based on a simplified version of their eutectic model II. Specifically, the dimensionless, symmetrical version of the model is used, with the additional assumptions that the system is isothermal and solid at all times. This dramatically reduces the complexity of the problem by eliminating the need for a third order parameter, as well as thermal effects.

The free energy function is constructed by assuming that the pure α and β phases have ideal solution free energy densities and calculating the total system free energy by taking weighted contributions of each pure free energy based on the value of ψ. The bulk Helmholtz free energy density is given by [1]:

$$f(T,c,\psi) = h(\psi)f^{\alpha}(T,c) + (1-h(\psi))f^{\beta}(T,c) + \frac{1}{4}W_{F\psi}\psi^2(1-\psi)^2$$

where $h(\psi) = \psi^2(3-2\psi)$. Assuming that the phase diagram is symmetric about $c = \frac{1}{2}$, the α and β phases have the same melting point in their pure forms, and

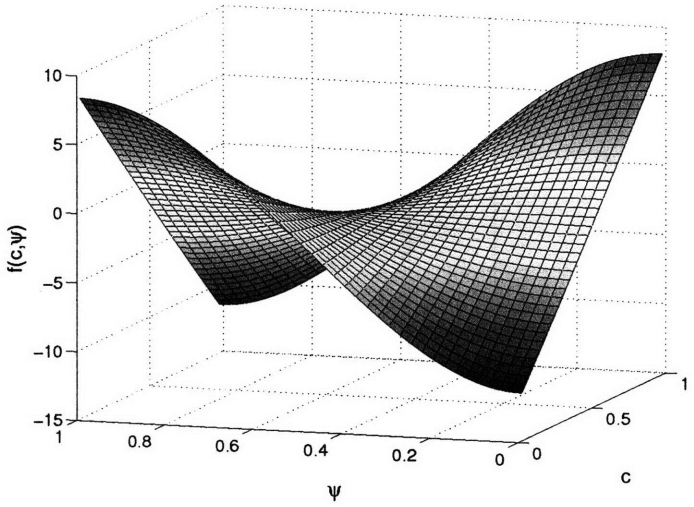


Figure 2-2: Free energy function described in Equation (2.4)

making all constants dimensionless yields [1]:

$$f(\tilde{T}_0, c, \psi) = \tilde{f}_0 + \frac{\tilde{W}_\psi}{4} g(\psi) + \tilde{T}_0 I(c) + \tilde{L}(\tilde{T}_0 - 1)[h(\psi)c + (1 - h(\psi))(1 - c)] + \tilde{L}(\tau\tilde{T}_0)[h(\psi)(1 - c) + (1 - h(\psi))c] \quad (2.4)$$

where $\tilde{T}_0$, $\tilde{W}_\psi$, $\tilde{f}_0$, and $\tilde{L}$ are dimensionless parameters, τ is a constant greater than 1 related to the degree of undercooling, $g(\psi) = \psi^2(1 - \psi)^2$, and $I(c) = c\ln(c) + (1 - c)\ln(1 - c)$.

A plot of the free energy function is included in Figure 2-2. It exhibits minima at $(\psi = 0, c = c_\alpha)$ and $(\psi = 1, c = c_\beta)$ as expected. The stable concentrations of the α and β phases can be determined by finding the minima of $f(c, \psi)$ when $\psi = 0$ and $\psi = 1$ respectively . The first derivative of $f(c, \psi)$ with respect to the concentration is given by:

$$\frac{\partial f}{\partial c} = \tilde{T}_0 \ln \frac{c}{1 - c} + [2h(\psi) - 1][\tilde{L}\tilde{T}_0(1 - \tau)]$$

setting this equal to 0 and solving yields:

$$c = \frac{e^{(1-2h(\psi))\tilde{L}(1-\tau)}}{1 + e^{(1-2h(\psi))\tilde{L}(1-\tau)}}$$

Given that $h(0) = 0$ and $h(1) = 1$:

$$c_\alpha = \frac{e^{\tilde{L}(1-\tau)}}{1 + e^{\tilde{L}(1-\tau)}} \tag{2.5}$$

$$c_\beta = \frac{e^{-\tilde{L}(1-\tau)}}{1 + e^{\tilde{L}(1-\tau)}} \tag{2.6}$$

2.2 Percolation Theory

The central focus of percolation theory is the study of cluster formations in a system, and the characterization of the properties of the clusters that exist in a given system. Traditionally most work on percolation has studied random percolation, and that case is sufficient to develop the basic theory that is then applied to more complex systems; therefore the remainder of this section will focus on random percolation. Specifically it will address random percolation on a two dimensional square lattice with a fraction p of occupied sites, that is every site on the lattice has probability p of being occupied and $1 - p$ of being unoccupied. Two sites on the lattice are considered to be part of the same cluster if and only if they are connected to one another by a path moving only on other occupied sites, and only from one site to its nearest neighbors [11]. A cluster is said to percolate if a path can be traced from one side of the system to the other while only touching sites that are part of that cluster.

The first quantity of interest in percolation theory is known as the percolation threshold, p_c. It is defined as the value of p such that a percolating cluster is never found in an infinite system for $p < p_c$, and at least one is always found for $p \geq p_c$ [11]. A number of other quantities diverge as $p \to p_c$, exhibiting similar behavior to many material properties around thermal phase transitions [11].

2.2.1 Critical Exponents and Universality

The behavior of many quantities that diverge near the percolation threshold can be modelled with a simple power law of the form:

$$X \propto |p - p_c|^b \tag{2.7}$$

where X is the quantity of interest, and b is a critical exponent [8]. One such quantity is the mean cluster size, S, which is defined as:

$$S = \frac{\sum_s s^2 n_s}{\sum_s n_s} \tag{2.8}$$

where n_s is the number of clusters of size s per lattice site [8]. It is logical that the mean cluster size grows rapidly as p approaches p_c, since at p_c there is at least one infinite cluster, therefore very slightly below p_c there should be a number of very large, but finite clusters, and the farther below p_c the smaller the clusters in the system are likely to be. As $p \rightarrow p_c$, S scales according to [11]:

$$S \propto |p - p_c|^{-\gamma} \tag{2.9}$$

The second such quantity of interest in this work is the strength of the infinite (or largest for $p < p_c$) cluster, P. P is defined as the fraction of sites belonging to the largest cluster in the system and approaches 0 as $p \rightarrow p_c$ according to the power law [11]:

$$P \propto |p - p_c|^{\beta} \tag{2.10}$$

Possibly the most interesting thing about the critical exponents is the phenomenon of universality. Previous studies suggest that the critical exponents depend solely on the dimensionality of the system in question [11]. For example, the same critical exponents are observed for square, triangular, and honeycomb lattices in two dimensions. Additionally, work done on continuum percolation suggests that two dimensional models of randomly placed discs belong to the same universality class as random

lattice percolation [2]. Therefore one of the main goals of this work is to determine whether diffusionally evolved systems also belong to the same universality class, or if there is a change in the underlying physics that results in different values for the critical exponents.

2.2.2 Finite Size Scaling

Thus far the question of how to obtain reliable estimates for p_c, as well as the critical exponents has not been addressed. The definitions in the previous section are all applicable to systems of infinite size; unfortunately it is impossible to simulate a system of infinite size in order to determine p_c, β, and γ. In order to accurately estimate these values one must investigate their scaling with system size, L.

In order to study critical behavior around the percolation threshold an accurate estimate of p_c is essential. Therefore the first question that must be addressed is how to determine p_c from simulations of finite systems. To accomplish this consider the probability of finding a spanning cluster in a system, Π. For an infinite system $\Pi = 1$ for $p \geq p_c$, and $\Pi = 0$ for $p < p_c$. However in a finite system it is easy to imagine a spanning cluster existing for values of p that are significantly lower than p_c, and similarly that it is possible for a spanning cluster to not be present for $p > p_c$. In general $\Pi(p, L)$ is expected be close to 0 for small values of p, and increase to approach 1 for large values of p, with the width of this transition being a function of L. As $L \rightarrow \infty$ a plot of $\Pi(p, L)$ will approach a step function at p_c [11]. An example plot of Π vs. p for two systems with $L < \infty$ and one at $L = \infty$ is presented in Figure 2-3 to illustrate the convergence of $\Pi(p, L)$ to a step function at p_c. Now define an effective percolation threshold p_c^{eff} as the value of p where $\Pi = \frac{1}{2}$ for a given L (note that the specific value of Π used to define p_c^{eff} is unimportant, any value between 0 and 1 is sufficient). Since Π approaches a step function as $L \rightarrow \infty$ it is necessary that $p_c^{\text{eff}} \rightarrow p_c$. It turns out that p_c^{eff} scales with system size according to:

$$p_c^{\text{eff}} - p_c \propto L^{\frac{-1}{\nu}} \tag{2.11}$$

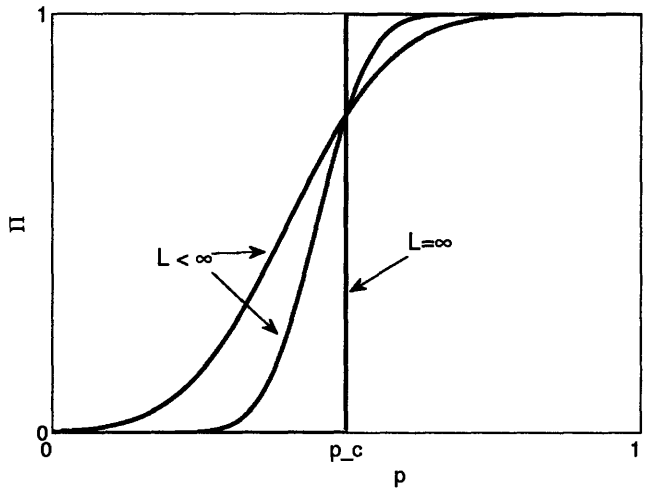


Figure 2-3: Example scaling of $\Pi(p, L)$ as system size increases to ∞.

where ν is a third critical exponent [11]. Unfortunately, without prior knowledge of ν this relation cannot be used directly to estimate p_c.

Now consider how $\Pi(p, L)$ approaches a step function as L is increased. Specifically, consider the scaling of the percolation transition width, Δ, with L. As with p_c^{eff} the specific definition of Δ used is unimportant as the scaling behavior remains the same, so for example the width of p that it takes for Π to increase from 0.1 to 0.9, or 0.2 to 0.8, or some other convenient measure of the transition width may be used. The scaling behavior of $\Delta(L)$ is described by [11]:

$$\Delta(L) \propto L^{\frac{-1}{\nu}} \tag{2.12}$$

Since both $\Delta(L)$ and p_c^{eff} scale according to power laws with the same critical exponent the following relation is used to estimate p_c:

$$p_c^{\text{eff}}(L) - p_c \propto \Delta(L) \tag{2.13}$$

p_c is determined by extrapolating the value of p_c^{eff} to $\Delta(L) = 0$, which corresponds to the thermodynamic limit of Π being a step function [11].

Similar finite size scaling techniques can be applied to $P(p, L)$ and $S(p, L)$ to determine the values of the critical exponents related to each based on simulations at

various L. Specifically the scaling of P and S with L at $p = p_c$ is given by [8]:

$$P(p_c, L) \propto L^{-\frac{\beta}{\nu}} \tag{2.14}$$

$$S(p_c, L) \propto L^{\frac{\gamma}{\nu}} \tag{2.15}$$

therefore β and γ can be determined from double logarithmic plots of $P(p_c, L)$ and $S(p_c, L)$ respectively.

2.2.3 Effects of Physical Correlations

Relatively little work has been done to study percolation in cases where there are correlations between particle's locations. However, many physical systems exhibit significant physical correlations due to diffusional effects, attractive or repulsive potentials between particles, and a variety of other factors. Therefore it is important to consider how these correlations could affect the percolation threshold and critical scaling exponents.

For example, consider a system of discs with uniform radius that interact through a Lennard-Jones potential of the form:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^1 2 - \left(\frac{\sigma}{r}\right)^6 \right] \tag{2.16}$$

where r is the radius between particles and σ controls the location of the potential minimum, r_0 [5]. If r_0 is larger than the disc radius then the discs will resist overlapping. This should lead to a noticeable increase in the percolation threshold relative to randomly placed discs. However, if r_0 is less than the disc radius the particles are more likely to coalesce into clusters. This should lower the percolation threshold. The presence of short range correlations like the Lennard-Jones potential can alter a system's p_c, however the system still belongs to the same universality class as random systems because the correlations disappear when the system is viewed on a sufficiently large length scale. In order for a system to belong to a different universality class, and therefore exhibit different critical exponents there must be some long range correla-

tions that have not previously been considered. This makes the observed difference between the critical exponents in this work and the expected values unexpected and exciting.

Chapter 3

Simulation Methods

3.1 Phase Field Method

3.1.1 Time Evolution

In order to use the phase field method to produce useful simulations it is necessary to discretize the governing equations in both time and space so that they may be solved numerically. For the purposes of this discussion the case of spinodal decomposition will be addressed. The methodology is the same for nucleation and growth, but the algebra is more complicated and both order parameters are updated based on their respective governing equations at each time step.

By differentiating Equation (2.3) and plugging the result into the Cahn-Hilliard equation (2.1), we obtain:

$$\frac{\partial c}{\partial t} = M_c \left\{ \nabla^2 \left[\frac{32 F_{max}}{(c_\beta - c_\alpha)^4} (c - c_\alpha)(c - c_\beta)(2c - c_\beta - c_\alpha) \right] - \epsilon_c^2 \nabla^4 c \right\}$$

$$\frac{\partial c}{\partial t} = M_c \frac{32 F_{max}}{(c_\beta - c_\alpha)^4} \left[2 \nabla^2 c^3 - 3(c_\alpha + c_\beta) \nabla^2 c^2 + (c_\alpha^2 + 4 c_\alpha c_\beta + c_\beta^2) \nabla^2 c \right] - M_c \epsilon_c^2 \nabla^4 c$$

First, we treat the space discretization by letting c be a vector defined at points on a grid x_i, with $i = 1 \ldots N$, and $\Delta x = x_{i+1} - x_i$. The Laplacian operator at x_i is then approximated by:

$$\left. \frac{d^2 c}{dx^2} \right|_{x_i} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2}$$

Assuming periodic boundary conditions, this can be applied to the entire vector c as a matrix operator:

$$\frac{d^2c}{dx^2} \approx \frac{1}{\Delta x^2} \begin{bmatrix} -2 & 1 & 0 & . & . & 1 \\ 1 & -2 & 1 & 0 & . & . \\ 0 & 1 & -2 & 1 & . & . \\ . & . & . & . & . & . \\ 1 & . & . & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_N \end{bmatrix}$$

For shorthand:

$$\frac{d^2c}{dx^2} \approx \frac{1}{\Delta x^2} D2\, c$$

where $D2$ is the matrix operator. This method can be extended to calculate the Laplacian in two dimensions by using $D2 = D2 + D2'$.

The time discretization is performed by approximating:

$$\left.\frac{\partial c}{\partial t}\right|_{t_n} \approx \frac{c_{n+1} - c_n}{\Delta t}$$

Applying these to the time evolution equation yields:

$$\begin{aligned} \frac{c_{n+1} - c_n}{\Delta t} =& M_c \frac{32 F_{max}}{(c_\beta - c_\alpha)^4} \left[2\frac{D2}{\Delta x^2} c_n^3 - 3(c_\alpha + c_\beta)\frac{D2}{\Delta x^2} c_n^2 + (c_\alpha^2 + 4c_\alpha c_\beta + c_\beta^2)\frac{D2}{\Delta x^2} c_n \right] \\ & - M_c \epsilon_c^2 \frac{D2^2}{\Delta x^4} c_{n+1} \end{aligned}$$

The final term on the right hand side uses c_{n+1} instead of c_n to allow for larger stable values of Δt. Separating the c_{n+1} terms to the left hand side, and c_n terms to the right hand side gives:

$$\begin{aligned} &\left(1 + M_c \epsilon_c^2 \frac{\Delta t D2^2}{\Delta x^4}\right) c_{n+1} = \\ &c_n + M_c \Delta t \frac{32 F_{max}}{(c_\beta - c_\alpha)^4} \left[2\frac{D2}{\Delta x^2} c_n^3 - 3(c_\alpha + c_\beta)\frac{D2}{\Delta x^2} c_n^2 + (c_\alpha^2 + 4c_\alpha c_\beta + c_\beta^2)\frac{D2}{\Delta x^2} c_n \right] \end{aligned}$$

Let:

$$LHS = \left(1 + M_c \epsilon_c^2 \frac{\Delta t D2^2}{\Delta x^4}\right)$$

$$RHS = c_n + M_c \Delta t \frac{32 F_{max}}{(c_\beta - c_\alpha)^4}\left[2\frac{D2}{\Delta x^2}c_n^3 - 1(c_\alpha + c_\beta)\frac{D2}{\Delta x^2}c_n^2 + (c_\alpha^2 + 4c_\alpha c_\beta + c_\beta^2)\frac{D2}{\Delta x^2}c_n\right]$$

$$c_{n+1} = LHS^{-1} RHS$$

Solving this equation to determine the new value of c at every time step would require inverting the matrix LHS, which can be computationally intensive. In order to reduce the computational intensity a Discrete Fourier Transform (DFT) is used to solve the matrix equation.

The DFT operates on a series of complex numbers $x_0 \ldots x_N$ transforming them to another series of complex numbers $X_0 \ldots X_N$ by [12]:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i}{N} kn} \quad k = 0 \ldots N-1 \tag{3.1}$$

Applying the DFT to a vector x yields a second vector X, so the DFT can be considered a matrix such that $X = \mathcal{F}x$, where $\mathcal{F}$ has the form:

$$\begin{bmatrix} \omega_N^{0*0} & \omega_N^{0*1} & \cdots & \omega_N^{0*(N-1)} \\ \omega_N^{0*1} & \omega_N^{1*1} & \cdots & \omega_N^{1*(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{0*(N-1)} & \omega_N^{1*(N-1)} & \cdots & \omega_N^{(N-1)(N-1)} \end{bmatrix}$$

$$\omega_N = e^{\frac{-2\pi i}{N}}$$

Therefore the k^{th} row of $\mathcal{F}$ is $\mathcal{F}_k = \left[1, e^{\frac{-2\pi i}{N}k}, e^{\frac{-2\pi i}{N}2k}, \ldots, e^{\frac{-2\pi i}{N}(N-1)k}\right]$. The transpose of the vectors $\mathcal{F}_k$ are eigenvalues of the $D2$ Laplacian operator such that $D2\ \mathcal{F}_k^T = \lambda_k \mathcal{F}_k^T$ and $\lambda_k = 2\cos k - 2$. Since $D2$ is symmetric $\mathcal{F}_k\ D2 = \lambda_k \mathcal{F}_k$ and $\mathcal{F}\ D2 = \Lambda \mathcal{F}$ where Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_k.

Now apply the DFT to the equation for the time evolution of c. First the left

hand side:

$$\mathcal{F}\left(1+M_c\epsilon_c^2\frac{\Delta t\ D2^2}{\Delta x^4}\right)c_{n+1}$$
$$\left(1+M_c\epsilon_c^2\frac{\Delta t\Lambda^2}{\Delta x^4}\right)\mathcal{F}c_{n+1}$$
$$\left(1+M_c\epsilon_c^2\frac{\Delta t\Lambda^2}{\Delta x^4}\right)\widehat{c_{n+1}}$$
$$\widehat{c_{n+1}}=\mathcal{F}c_{n+1}$$

Next do the same to the right hand side:

$$\mathcal{F}\left\{c_n+M_c\Delta t\frac{32F_{max}}{(c_\beta-c_\alpha)^4}\left[2\frac{D2}{\Delta x^2}c_n^3-3(c_\alpha+c_\beta)\frac{D2}{\Delta x^2}c_n^2+(c_\alpha^2+4c_\alpha c_\beta+c_\beta^2)\frac{D2}{\Delta x^2}c_n\right]\right\}$$
$$\widehat{c_n}+M_c\Delta t\frac{32F_{max}}{(c_\beta-c_\alpha)^4}\left[2\frac{\Lambda}{\Delta x^2}\widehat{c_n^3}-3(c_\alpha+c_\beta)\frac{\Lambda}{\Delta x^2}\widehat{c_n^2}+(c_\alpha^2+4c_\alpha c_\beta+c_\beta^2)\frac{\Lambda}{\Delta x^2}\widehat{c_n}\right]$$

Solving:

$$\widehat{c_{n+1}}=\left(1+M_c\epsilon_c^2\frac{\Delta t\ \Lambda^2}{\Delta x^4}\right)^{-1}$$
$$\left\{\widehat{c_n}+M_c\Delta t\frac{32F_{max}}{(c_\beta-c_\alpha)^4}\left[2\frac{\Lambda}{\Delta x^2}\widehat{c_n^3}-3(c_\alpha+c_\beta)\frac{\Lambda}{\Delta x^2}\widehat{c_n^2}+(c_\alpha^2+4c_\alpha c_\beta+c_\beta^2)\frac{\Lambda}{\Delta x^2}\widehat{c_n}\right]\right\}$$

Since Λ is a diagonal matrix this can be separated into k separate equations each of which is easily solved, eliminating the need for solving a large matrix equation. Once $\widehat{c_{n+1}}$ is computed c_{n+1} is computed using the inverse DFT, $c_{n+1}=\mathcal{F}^{-1}\widehat{c_{n+1}}$.

The simplest extension of this method to two dimensions involves adding $D2$ to its transpose matrix to account for the additional dimension. This matrix has eigenvalues:

$$\lambda_{j,k}=2(\cos\frac{2\pi j}{N_x}+\cos\frac{2\pi k}{N_y}-2)$$

However using this finite difference approximation ignores contributions at 45° angles to the axes, and results in visible anisotropy in simulation results. To eliminate this

anisotropy a filter of the form:

$$\begin{matrix} \alpha & \gamma & \alpha \\ \gamma & -4(\alpha+\gamma) & \gamma \\ \alpha & \gamma & \alpha \end{matrix}$$

is used [3]. The finite difference matrix for this filter has eigenvalues:

$$\lambda_{j,k} = \frac{2}{3}\left(cos(\frac{2\pi j}{N_x}) * cos(\frac{2\pi k}{N_y}) - 1\right) + \frac{4}{3}\left(cos(\frac{2\pi j}{N_x}) + cos(\frac{2\pi k}{N_y}) - 2\right)$$

The functions that control the time evolution of the system for spinodal decomposition and nucleation and growth are binary_alloy3D_fft_chonly(), and binary_alloy3D_fft() respectively. Both are included in their entirety in the file phasefield3D.cpp in Appendix A.

3.1.2 Initial Conditions

Now that we have addressed the method for solving the time evolution equations the initial conditions for the simulations must be specified. For the case of spinodal decomposition the value of the concentration matrix at each grid point is set to $c_{i,j} = c_0 \pm \delta$ where c_0 is the average initial concentration determined by the desired final phase fraction of the β phase, and δ is a random number on the order of 10^{-3} added so that there is some noise in the system. The value of c_0 is computed for each simulation using the lever rule and the desired final phase fraction of β, p_∞ . It is given by:

$$c_0 = c_\alpha + p_\infty(c_\beta - c_\alpha) \tag{3.2}$$

where c_α and c_β are the stable concentrations of the α and β phases. The code performing this initialization is in the function fftgrid3D::initializeValueWithNoise() in the file fftgrid3D.cpp in Appendix A.

The initial conditions for the nucleation and growth simulations are somewhat more complicated. Rather than starting with the entire system centered around

a given concentration with some added noise, it begins as mainly α phase, with randomly placed nuclei of the β phase. In order to ensure that the system reaches the desired fraction of β phase at equilibrium, the average initial concentration of the system must still be c_0 as described in Equation (3.2). In order to accomplish this a second parameter is used in the initialization of the nucleation and growth simulations, p_0^∞. It determines the initial fraction of the system that is covered by nuclei of the β phase according to $p_0 = p_0^\infty p_\infty$.

The algorithm for initializing the system to meet these criteria is as follows. First the number of nuclei to be placed is determined based on the calculated value of p_0. It is given by:

$$N_{nuclei} = \frac{p_0 N_x N_y}{s}$$

where N_x and N_y are the dimensions of the system and s is the size of each individual nucleus. Now, knowing the desired average concentration of the system the concentration of component B outside of the nuclei can be calculated:

$$c_{else} = \frac{N_x N_y c_0 - N_{nuclei} s c_\beta}{N_x N_y - N_{nuclei} s}$$

Once this is computed the value of c at every grid point is set to c_{else} and ψ is set to 0.01 at every grid point. The nuclei are then placed by selecting a random grid location as the center, checking that there would be no overlap with nearby nuclei, and setting $c_{i,j} = c_\beta$ and $\psi_{i,j} = 0.99$ for all grid locations with radius r of the center location. This process is repeated until all N_{nuclei} are placed. The code used to perform this initialization is in the function fftgrid3D::initializeNuclei(), in the file fftgrid3D.cpp in Appendix A

3.1.3 Parameter Values

A number of dimensionless parameters are used in the free energy models for both spinodal decomposition and nucleation and growth. Additional parameters governing the kinetics of the system are included in the Cahn-Hilliard and Allen-Cahn equations.

The values of all parameters used in this work are listed in Table 3.1.

Table 3.1: Model Parameters for Spinodal Decomposition and Nucleation and Growth

	Spinodal Decomposition	Nucleation and Growth
dt	0.00001	0.001
M_c	0.001	0.0001
M_ψ	N/A	1.0
F^{max}	1.0	N/A
c_α	0.881	N/A
c_β	0.119	N/A
ϵ_c	0.015	0.005
ϵ_ψ	N/A	0.005
$\tilde{W}_\psi$	N/A	10.0
$\tilde{L}$	N/A	20.0
$\tilde{T}_0$	N/A	0.4
τ	N/A	1.1

3.2 Cluster Counting

In order to analyze the data from the phase field simulations and study percolation behavior, it is necessary to have an efficient algorithm for marking clusters in the system. The primary difficulty in finding such an algorithm is easily illustrated with a simple example. Consider the lattice depicted below, where an x signifies an occupied space:

$$\begin{array}{ccccc} x & & x & & x \\ x & x & x & & x \\ x & x & x & x & x \end{array}$$

Traversing the array from left to right, then top to bottom the first site we encounter is occupied so is labelled 1, the next site is unoccupied so is labelled 0, the third site is occupied and gets labelled 2, the fourth site is labelled 0, and the fifth 3. No problems so far. On the second line the first two sites are occupied and connected to the cluster labelled 1 on the first line, so they are both labelled 1, but now at the third site of the second line there is a problem. This site is connected on the left to

the cluster labelled 1, and above to the cluster labelled 2.

$$\begin{array}{ccccc} 1 & & 2 & & 3 \\ 1 & 1 & ? & & x \\ x & x & x & x & x \end{array}$$

The most important part of a cluster counting algorithm is how it deals with this case. It is obvious that the clusters previously labelled 1 and 2 are in fact a single cluster so how should all of the sites labelled 2 be relabelled? A similar problem arises when the algorithm reaches the bottom right site of this example matrix as cluster 3 now joins the single large cluster. A naive algorithm would simply start over from the beginning and relabel every 2 as a 1 when it first encounters the situation, but that is extremely inefficient. One solution to this problem which allows for counting clusters in $O(N)$ time instead of the $O(N^2)$ time that the naive algorithm takes is the Hoshen-Kopelmann algorithm [11].

The Hoshen-Kopelmann algorithm works by maintaining a list of incorrect labels along with what label each "bad" label should have. This is managed by maintaining an array whose length is the number of labels in use, call it L. Each $L(i)$ stores the correct label for label i, thus if $L(i) = i$ i is a proper label. Each time two previously marked clusters are found to connnect L is updated to reflect that. In the example above when the algorithm reaches the third site in the second row it would label that site 1 and set $L(2) = 1$. Then when it reached the final site in the lattice it would label it 1 and set $L(3) = 1$. Then a second traversal of the array is sufficient to label each site properly by relabelling based on the values of $L(i)$.

Once the system is labelled, it is trivial to determine whether or not a spanning cluster is present. Simply compare the list of clusters present in the top row to those in the bottom row, and if a cluster appears in both lists then it is a spanning cluster. For the simulations in this work only systems with clusters spanning both top to bottom and left to right were considered to have percolated. To confirm whether or not a cluster spanning top to bottom also spans left to right simply check whether any sites in the leftmost and rightmost columns are occupied with that cluster label. The code

implementing the Hoshen-Kopelmann algorithm is included in the file fftgrid3D.cpp in Appendix A.

An alternative, and more elegant, solution to marking clusters also exists. This recursive solution marks an entire cluster at a time, and by repeatedly applying it until no unlabelled occupied sites exist every cluster can be marked while only visiting each site in the lattice once. The recursive method for labelling a cluster is as follows. First check that the current site is occupied and has not yet been labelled. If it is unoccupied then return. If it is occupied then label it and proceed to call the function with each of its neighbors as the site to be checked. Pseudocode for a function that uses this algorithm is below, although it doesn't account for boundary conditions.

```
void burnandlight(int **data, int i, int j, int label) {
  if(data[i][j] != 1)
    //this site is either unoccupied, or has already been marked with
    //this label
    return;
  //label this site
  data[i][j] = label;
  //mark all neighboring sites
  burnandlight(data,i+1,j,label);
  burnandlight(data,i-1,j,label);
  burnandlight(data,i,j+1,label);
  burnandlight(data,i,j-1,label);
}
```

While this algorithm should have the same algorithmic complexity as the Hoshen-Kopelmann algorithm some scaling problems can occur due to its recursive nature. Every additional site in a cluster leads to four additional function calls and their associated memory overhead. Figure 3-1 plots the time required to mark the clusters in a system versus the fraction of unoccupied sites. In order to avoid any potential scaling problems the Hoshen-Kopelmann method was used during all simulations in this work.

The other relevant statistics used in this work are the power of the infinite, or in the case of finite size systems, the largest cluster, $P(p, L)$ and the square of the mean

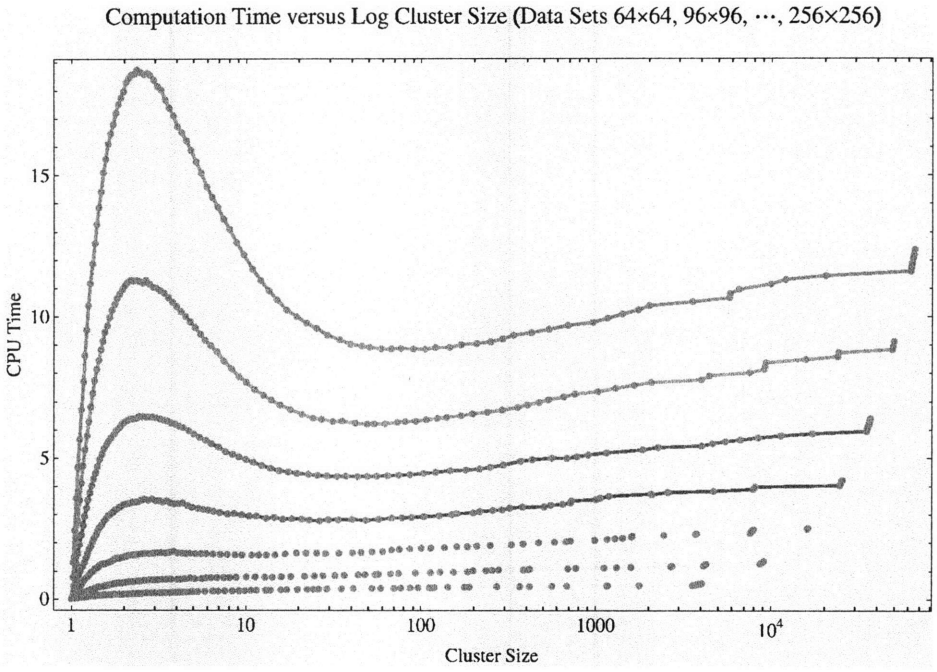


Figure 3-1: CPU time to mark clusters vs. fraction of unoccupied sites for systems of size L=64, 96, ..., 256

cluster size, $S(p, L)$. To calculate these values for a system that has had its clusters marked an array N of length L, where L is the number of unique clusters, is created. $N(i)$ stores the number of sites in the cluster labelled i, and is calculated by traversing the system and incrementing $N(i)$ every time a site labelled i is encountered. The value of $P(p, L)$ is then given by $\frac{max(N(i))}{L^2}$, and $S(p, L)$ is given by:

$$S = \frac{1}{L-1} \sum_{i=1}^{L-1} N(i)^2$$

this assumes that $N(i)$ is sorted in ascending order so that the size of the largest cluster is stored in $N(L)$ and is not included in the sum.

3.3 Methodology

In order to accurately estimate p_c and the critical exponents a large number of simulations must be run for various values of p and L. The first step is to estimate p_c by generating $\Pi(p, L)$ curves for several different system sizes. For both spinodal decomposi-

tion and nucleation and growth system sizes of $L = 64, 96, 128, 192, 256, 384,$ and 512. For each system size simulations were run at values of p_∞ spanning the percolation transition in increments of $\Delta p_\infty = 0.01$. In order to obtain a useful estimate of $\Pi(p, L)$ for each data point between 60 and 250 simulations were performed, with larger systems having fewer simulations per point due to the longer time for a simulation to run to completion. Upon calculation of p_c from this data additional sets of 90 to 250 simulations were run at $p = p_c$ and the same range of L to provide the data to calculate β and γ.

Chapter 4

Results and Discussion

4.1 Simulations of Microstructural Evolution with Combined Growth and Coarsening.

Time lapse images of two sample systems are presented in Figure 4-1. Parts (a) through (e) depict the evolution of a spinodally decomposed system with $p_\infty = 0.5$. The system starts at a uniform, unstable composition at $t = 0$ then decomposes into two phases with compositions c_α and c_β in (b) - (d). As expected for spinodal decomposition the width of each phase exhibits a characteristic length scale [9]. Some coarsening of the microstructure is evident between (d) and (e). Parts (f) - (j) show the evolution of a system that underwent nucleation and growth with $p_\infty = 0.63$. Critical nuclei are visible in (f). These nuclei draw in solute from the surrounding area to grow, creating a depletion region that is visible during the initial growth phase, as in (g). After the initial growth phase the microstructure coarsens, as seen from (h) - (j).

4.2 Percolation Threshold

Figure 4.2 contains plots of the percolation probability vs. equilibrium volume fraction of β phase for systems of various sizes evolved by spinodal decomposition and

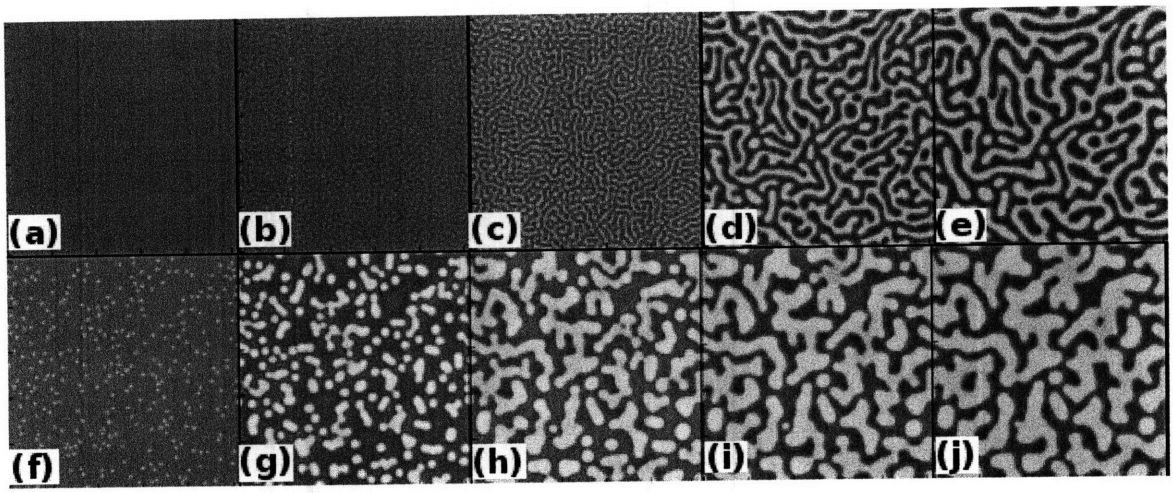
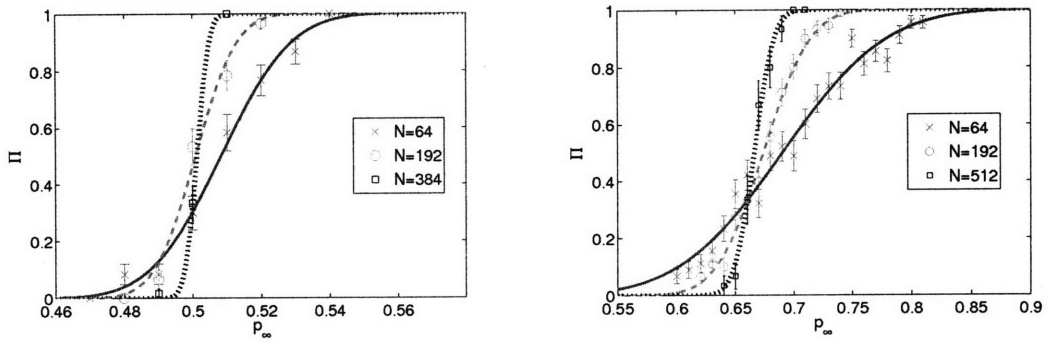


Figure 4-1: Sample simulation results for spinodal decomposition (a-e) and nucleation and growth (f-j). The spinodal decomposition images are for a system with $p_\infty = 0.5$, taken at t=0, 750, 1500, 10000, and 20000 time steps. The nucleation and growth images are for a system with $p_\infty = 0.63$ taken at t=0, 5000, 10000, 15000, and 20000 time steps.

nucleation and growth. The decrease in percolation transition width as system size increases is quite apparent from these plots. By fitting each set of data with the empirical equation:

$$\frac{1 + erf\left[\frac{p - p_c^{\text{eff}}}{\Delta}\right]}{2}$$

we obtain values for $p_c^{\text{eff}}(L)$ and $\Delta(L)$, allowing p_c to be estimated using Equation (2.13).

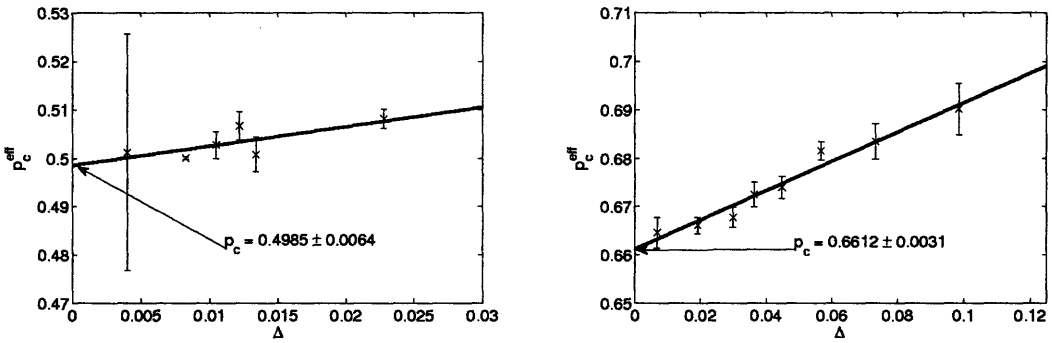


(a) Spinodal Decomposition

(b) Nucleation and Growth

Figure 4-2: Plot of the probability of percolation Π versus equilibrium phase fraction of β phase, p_∞. Curves are plotted for multiple system sizes to illustrate the shrinking transition width as the size of the system is increased.

The fitted values of $p_c^{\text{eff}}(L)$ and $\Delta(L)$ are plotted in Figure 4-3. The value of p_c in the infinite limit of $\Delta = 0$ is extrapolated from the y intercept of a linear regression on this data. This yields $p_c = 0.4985 \pm 0.0064$ for spinodally decomposed systems, and $p_c = 0.6612 \pm 0.0031$ for systems that underwent nucleation and growth. In comparison the percolation threshold for a randomly assigned square lattice is



(a) Spinodal Decomposition

(b) Nucleation and Growth

Figure 4-3: Plot of the effective percolation threshold, p_c^{eff} versus the width of the percolation transition Δ.

$p_c = 0.592746$ [11]. Continuum percolation of overlapping discs exhibits a threshold $p_c = 0.6764 \pm 0.0009$ when the discs have a uniform radius, and $p_c = 0.6860 \pm 0.0012$ for discs with a distribution of radii [2].

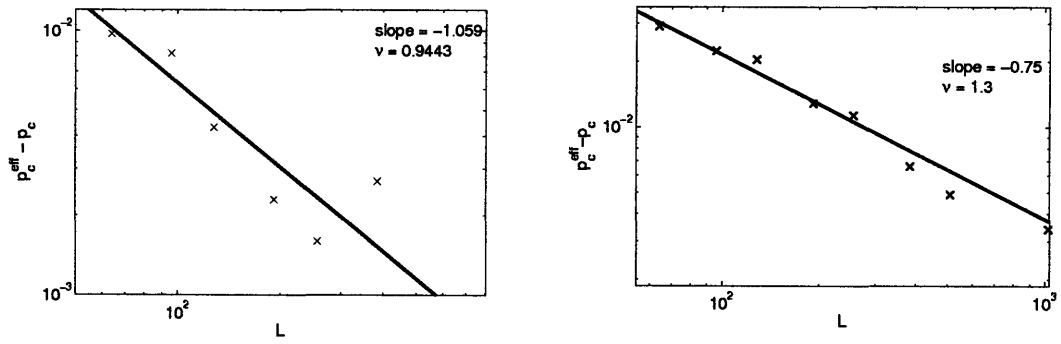
Due to the nature of spinodal decomposition it is unsurprising that it exhibits the lowest percolation threshold of all the mentioned cases. Spinodally decomposed microstructures are self similar, and tend to form highly connected networks. Therefore a spanning cluster is more likely to form at a lower phase fraction than for a random square lattice, or randomly distributed discs. The observed percolation threshold for systems evolved through nucleation and growth is significantly higher than that of both spinodal decomposition and the random square lattice. This can be explained by the presence of the depletion region that forms around a nucleus as it grows. As two growing nuclei get closer to one another their depletion regions will impinge, and the lack of solute slows the growth of the nuclei towards one another therefore reducing the chance that they coalesce into a single particle. The observed decrease in percolation threshold compared to the continuum case of randomly placed discs

can be explained by a driving force for nearby nuclei to coalesce that is not present for random placement. This driving force is due to the reduction in total surface area that occurs when nearby nuclei join to form a single larger particle.

4.3 Critical Exponents

4.3.1 ν Estimation

The value of the critical exponent ν is estimated based on the scaling behavior of p_c^{eff} with system size as described in Equation (2.11). Since p_c^{eff} exhibits power law scaling with system size the value of ν is calculated from the slope of the best fit to a double logarithmic plot of the data. This is shown in Figure 4-4. From these data we obtain an estimate $\nu = 0.9 \pm 0.5$ for spinodal decomposition, and $\nu = 1.3 \pm 0.2$ for nucleation and growth. The value for nucleation and growth does not differ significantly from the exact value $\nu = 4/3$ [11]. However, the estimate for spinodal decomposition is quite different than expected with 4/3 falling at the far edge of the error range.



(a) Spinodal Decomposition

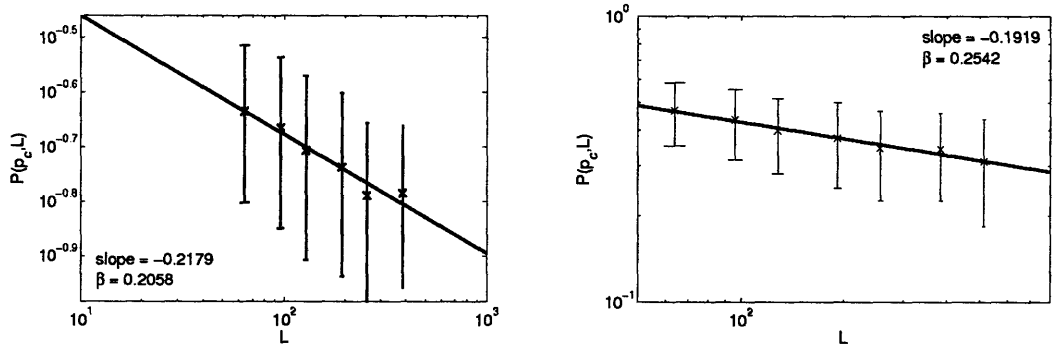
(b) Nucleation and Growth

Figure 4-4: Graphical determination of the exponent ν from the log-log plot of $p_c^{\text{eff}} - p_c$ versus the system size L.

4.3.2 β Estimation

β is estimated from the scaling behavior of the strength of the infinite cluster at the percolation threshold, $P(p_c, L)$. For finite systems the fraction of sites in the

largest cluster is used since there is no infinite cluster. According to Equation(2.14), $P(p_c, L)$ follows a power law, so β is estimated from the slope of a double logarithmic plot of $P(p_c, L)$ vs. L. This plot is presented in Figure 4-5. Since the slope, m, of the best fit line should equal the exponent of the power law $\beta = -m\nu$. This yields $\frac{\beta}{\nu} = 0.218 \pm 0.063$ spinodal decomposition and $\frac{\beta}{\nu} = 0.192 \pm 0.033$ for nucleation and growth. Using the estimates of ν obtained in Section 4.3.1 gives $\beta = 0.21 \pm 0.13$ for spinodal decomposition , and $\beta = 0.250 \pm 0.058$ for nucleation and growth. Both results differ from the expected value of $\beta = \frac{5}{36} = 0.1389$ for two dimensional systems, and the difference is statistically significant at a 95% confidence level for the case of nucleation and growth.[11]. This suggests that both spinodal decomposition and nucleation and growth belong to a different universality class than all previously addressed two dimensional percolation problems.



(a) Spinodal Decomposition

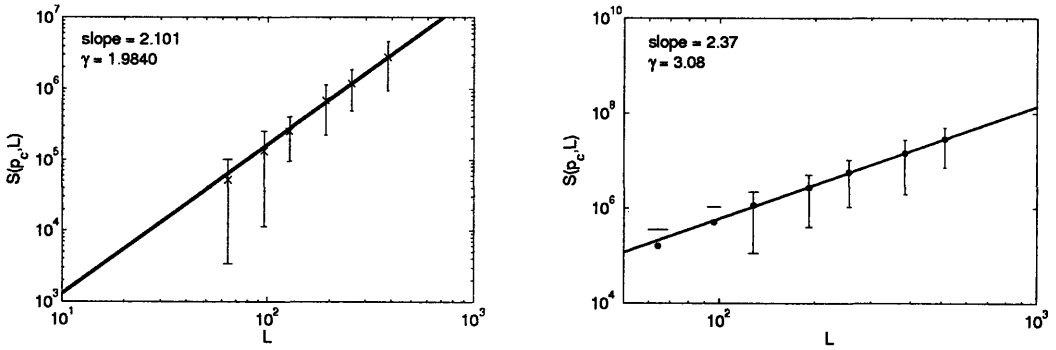
(b) Nucleation and Growth

Figure 4-5: Graphical determination of the exponent β from the log-log plot of $P(p_c, L)$ versus the system size L.

4.3.3 γ Estimation

Similarly to ν and β, the value of γ is best estimated from a double logarithmic plot of a quantity that exhibits power law scaling with system size. That quantity is the mean cluster size $S(p_c, L)$. According to Equation (2.15), the slope of the double logarithmic plot should be $\frac{\gamma}{\nu}$, therefore $\gamma = m\nu$. From the data plotted in Figure 4-6 we obtain $\frac{\gamma}{\nu} = 2.101 \pm 0.093$ for spinodal decomposition, and $\frac{\gamma}{\nu} = 2.368 \pm 0.049$ for nucleation and growth. Using the estimates of ν from Section 4.3.1 to solve for γ

gives $\gamma = 2.0 \pm 1.1$ for spinodal decomposition and $\gamma = 3.08 \pm 0.48$ for nucleation and growth. The expected value for a two dimensional system is $\gamma = \frac{43}{18} = 2.3889$. As was the case with the estimates of β for both spinodal decomposition and nucleation and growth, the estimates of γ for both differ from the expected value, with the difference being statistically significant for the case of nucleation and growth. This provides further evidence that the underlying physics of these cases is more complex than for the simpler random percolation problems that were studied previously.



(a) Spinodal Decomposition

(b) Nucleation and Growth

Figure 4-6: Graphical determination of the exponent γ from the log-log plot of $S(p_c, L)$ versus the system size L.

Chapter 5

Conclusion

Physical phase correlations introduced as a result of diffusion moderated microstructural evolution appear to significantly alter the underlying physics of percolation in comparison to other two dimensional systems. The shifted values of p_c for spinodal decomposition and nucleation and growth are logically explained based on the nature of their microstructural evolution. However, the observed differences in the critical exponents β and γ are somewhat surprising considering the previously posited phenomenon of universality. While the errors are relatively large, the major source is the error in the estimate of ν and the estimates of $\frac{\beta}{\nu}$ and $\frac{\gamma}{\nu}$ are more precise and still differ from the expected values for two dimensional systems. These results suggest that diffusionally evolved systems like those examined in this work belong to a different universality class than other two dimensional percolation problems.

Appendix A

Simulation Code

A.1 fftgrid3D.h

```
#ifndef _FFTGRID3D_H_
#define _FFTGRID3D_H_

#include <fstream>
#include <iostream>
#include <math.h>
#include <fftw3.h>
#include "sse2.h"
#include "sse3.h"
using namespace std;

//These macros are not used in the operators because having to do 2 multiplies
//and 2 adds to calculate the index is really slow.
#define gridLoop3D(grid) for (int i=0; i<(grid).getDimension(1); ++i) for (int j=0; j
    <(grid).getDimension(2); ++j) for (int k=0; k<(grid).getDimension(3); ++k)
#define rowmajindex k+N3*(j+N2*i)

#define X_DIM 1
#define Y_DIM 2
#define Z_DIM 3
#define XY_PLANE 8
#define XZ_PLANE 9
#define YZ_PLANE 10

class fftgrid3D{
  //define some operators
  inline friend fftgrid3D operator+(const fftgrid3D&,const fftgrid3D&);
```

```
27    inline friend fftgrid3D operator-(const fftgrid3D&,const fftgrid3D&);
      inline friend fftgrid3D operator*(const fftgrid3D&,const fftgrid3D&);
      inline friend fftgrid3D operator/(const fftgrid3D&,const fftgrid3D&);
      inline friend fftgrid3D operator*(const fftgrid3D&,const double);
      inline friend fftgrid3D operator*(const double, const fftgrid3D&);
32    inline friend fftgrid3D operator/(const fftgrid3D&,const double);
      inline friend fftgrid3D operator+(const fftgrid3D&,const double);
      inline friend fftgrid3D operator+(const double, const fftgrid3D&);
      inline friend fftgrid3D operator-(const fftgrid3D&,const double);
      inline friend fftgrid3D operator-(const double, const fftgrid3D&);
37    public:
      fftgrid3D(int,int,int, double=0);
      fftgrid3D(char*, double=0, int=0, int=0, int=0, int=0, int=0);
      fftgrid3D(const fftgrid3D&);
      ~fftgrid3D(void);
42    void initializeValueWithNoise(double);
      void initializeNuclei(double,double,double,int,fftgrid3D*);
      int hoshenKopelman(double);
      double volumeFraction(double);
      void writeToFile(char*);
47    void writeToFileComplex(char*);
      fftw_complex* getGrid(){return grid;};
      fftw_complex* getGrid()const{return grid;};
      inline int getDimension(int n){int dim=0; switch (n){case X_DIM: dim=N1; break;
          case Y_DIM: dim=N2; break; case Z_DIM: dim=N3; break; case 4: dim=N1_orig;
          break; case 5: dim=N2_orig; break; case 6: dim=N3_orig; break;}; return(dim);}
      inline double* operator()(int,int,int);
52    inline double* operator()(int,int,int) const; //for access only
      inline void operator=(const double);
      inline void operator=(const fftgrid3D&);
      inline void operator=(fftw_complex*);
      void naturallog();
57    fftgrid3D fft();
      fftgrid3D ifft();

     protected:
      void allocate(int,int,int);
62    int N1;
      int N2;
      int N3;
      fftw_plan plan_for;
      fftw_plan plan_rev;
67    fftw_complex* grid;
      double* grid_result;
      fftw_complex* grid_result2;
```

```
    static int fftw_init_threads_called;
    void init_fftw();
72  void cleanup_fftw();

    /* N_inc variables are used to keep track of the initial size of the grid
     * that was read from the input file.  warned_range and warned_bndry are used
     * to insure that particular warning messages are only displayed once to
77   * avoid filling output files with error messages.
     */
   private:
    int N1_orig;
    int N2_orig;
82  int N3_orig;
    int warned_range;
    int warned_bndry;
};

87 //-----------------------------------------------------------------------
void fftgrid3D::operator=(const double value){
    int max = N1*N2*N3;
    for(int i=0; i<max; ++i) {
      grid[i][0] = value;
92    grid[i][1] = 0;
    }
}
//-----------------------------------------------------------------------
void fftgrid3D::operator=(const fftgrid3D &rhs){
97  memcpy(grid,rhs.getGrid(),sizeof(fftw_complex)*N1*N2*N3);
}
//-----------------------------------------------------------------------
void fftgrid3D::operator=(fftw_complex *rhs){
    memcpy(grid,rhs,sizeof(fftw_complex)*N1*N2*N3);
102 }
//-----------------------------------------------------------------------
//This function provides access to elements in the grid
double* fftgrid3D::operator()(int i, int j, int k){
    return (double*)grid[rowmajindex];
107 }

double* fftgrid3D::operator()(int i, int j, int k) const {
    return (double*)grid[rowmajindex];
}
112

//This function adds two grids together
fftgrid3D operator+(const fftgrid3D &lhs,const fftgrid3D &rhs) {
```

```
    //should check that dimensions of this and rhs are the same
    fftgrid3D answer(lhs);
117 fftw_complex *lg, *rg, *ag;
    lg = lhs.getGrid();
    rg = rhs.getGrid();
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
122 for(int i=0;i<max;++i) {
#ifdef __SSE2__
      sseAdd((double*)lg[i], (double*)rg[i], (double*)ag[i]);
#else
      ag[i][0] = lg[i][0] + rg[i][0];
127   ag[i][1] = lg[i][1] + rg[i][1];
#endif
    }
    return answer;
  }
132
  //subtracts 2 grids
  fftgrid3D operator-(const fftgrid3D &lhs,const fftgrid3D &rhs) {
    //should check that dimensions of this and rhs are the same
    fftgrid3D answer(lhs);
137 fftw_complex *lg, *rg, *ag;
    lg = lhs.getGrid();
    rg = rhs.getGrid();
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
142 for(int i=0;i<max;++i) {
#ifdef __SSE2__
      sseSub((double*)lg[i], (double*)rg[i], (double*)ag[i]);
#else
      ag[i][0] = lg[i][0] - rg[i][0];
147   ag[i][1] = lg[i][1] - rg[i][1];
#endif
    }
    return answer;
  }
152
  //This function multiplies 2 grids element by element
  fftgrid3D operator*(const fftgrid3D &lhs,const fftgrid3D &rhs) {
    //should check dimensions
    fftgrid3D answer(lhs);
157 fftw_complex *lg, *rg, *ag;
```

```
    lg = lhs.getGrid();
    rg = rhs.getGrid();
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
162 for(int i=0;i<max;++i) {
#ifdef __SSE3__
      sse3ComplexMult((double*)lg[i], (double*)rg[i], (double*)ag[i]);
#elif defined(__SSE2__)
      sseComplexMult((double*)lg[i], (double*)rg[i], (double*)ag[i]);
167 #else
      ag[i][0] = lg[i][0]*rg[i][0]-lg[i][1]*rg[i][1];
      ag[i][1] = lg[i][0]*rg[i][1]-lg[i][1]*rg[i][0];
#endif
    }
172 return answer;
}

fftgrid3D operator/(const fftgrid3D &lhs,const fftgrid3D &rhs) {
    fftgrid3D answer(lhs);
177 fftw_complex *lg, *rg, *ag;
    lg = lhs.getGrid();
    rg = rhs.getGrid();
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
182 for(int i=0;i<max;++i) {
#ifdef __SSE3__
      sse3ComplexDiv((double*)lg[i], (double*)rg[i], (double*)ag[i]);
#elif defined(__SSE2__)
      sseComplexDiv((double*)lg[i], (double*)rg[i], (double*)ag[i]);
187 #else
      double a = lg[i][0];
      double b = lg[i][1];
      double c = rg[i][0];
      double d = rg[i][1];
192   double denom = c*c + d*d;
      ag[i][0] = (a*c + b*d)/denom;
      ag[i][1] = (b*c - a*d)/denom;
#endif
    }
197 return answer;
}

//These functions multiply every value in a grid by a constant
```

```
fftgrid3D operator*(const fftgrid3D &lhs,const double rhs) {
  fftgrid3D answer(lhs);
  fftw_complex *lg, *ag;
  lg = lhs.getGrid();
  ag = answer.getGrid();
  int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
      (Z_DIM);
  for(int i=0;i<max;++i) {
#ifdef __SSE2__
    sseComplexConstMult((double*)lg[i], rhs, (double*)ag[i]);
#else
    ag[i][0] = lg[i][0]*rhs;
    ag[i][1] = lg[i][1]*rhs;
#endif
  }
  return answer;
}

fftgrid3D operator*(const double lhs,const fftgrid3D &rhs) {
  return rhs*lhs;
}

fftgrid3D operator/(const fftgrid3D &lhs, const double rhs) {
  fftgrid3D answer(lhs);
  fftw_complex *lg, *ag;
  lg = lhs.getGrid();
  ag = answer.getGrid();
  int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
      (Z_DIM);
  for(int i=0;i<max;++i) {
#ifdef __SSE2__
    sseComplexConstDiv((double*)lg[i], rhs, (double*)ag[i]);
#else
    ag[i][0] = lg[i][0]/rhs;
    ag[i][1] = lg[i][1]/rhs;
#endif
  }
  return answer;
}

//These functions add a constant to every value in a grid
fftgrid3D operator+(const fftgrid3D &lhs, const double rhs) {
  fftgrid3D answer(lhs);
  fftw_complex *lg, *ag;
  lg = lhs.getGrid();
```

```
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
    for(int i=0;i<max;++i) {
247   ag[i][0] = lg[i][0]+rhs;
      ag[i][1] = lg[i][1];
    }
    return answer;
  }
252
  fftgrid3D operator+(const double lhs,const fftgrid3D &rhs) {
    return rhs+lhs;
  }

257 //These functions subtract a constant from every value in a grid
  fftgrid3D operator-(const fftgrid3D &lhs, const double rhs) {
    fftgrid3D answer(lhs);
    fftw_complex *lg, *ag;
    lg = lhs.getGrid();
262 ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
    for(int i=0;i<max;++i) {
      ag[i][0] = lg[i][0]-rhs;
      ag[i][1] = lg[i][1];
267 }
    return answer;
  }
  fftgrid3D operator-(const double lhs,const fftgrid3D &rhs) {
    fftgrid3D answer(rhs);
272 fftw_complex *rg, *ag;
    rg = rhs.getGrid();
    ag = answer.getGrid();
    int max = answer.getDimension(X_DIM)*answer.getDimension(Y_DIM)*answer.getDimension
        (Z_DIM);
    for(int i=0;i<max;++i) {
277   ag[i][0] = lhs - rg[i][0];
      ag[i][1] = -1*rg[i][1];
    }
    return answer;
  }
282
  #undef rowmajindex
  #endif
```

A.2 fftgrid3D.cpp

```
#include "fftgrid3D.h"

int fftgrid3D::fftw_init_threads_called = 0;

//Grid macros
//-----------------------------------------------------------------------
#define gridLoop for(int i=0; i<N1; ++i) for (int j=0; j<N2; ++j) for (int k=0; k<N3;
    ++k)
#define sq(x) ((x)*(x))
#define cube(x) ((x)*sq(x))
#define PI 3.14159265

//-----------------------------------------------------------------------
fftgrid3D::fftgrid3D(int n1, int n2, int n3, double initialVal) {
  if(!fftw_init_threads_called) {
    fftw_init_threads();
    fftw_plan_with_nthreads(2);
    fftw_init_threads_called++;
  }
  //Initialize default values for variables
  N1_orig=0;
  N2_orig=0;
  N3_orig=0;
  warned_range=0;
  warned_bndry=0;

  N1=n1;
  N2=n2;
  N3=n3;

  //Allocate space for the grid
  allocate(N1,N2,N3);
  grid_result = NULL;
  grid_result2 = NULL;
  plan_for = NULL;
  plan_rev = NULL;
  (*this)=initialVal;
}
//-----------------------------------------------------------------------
//This function allows a fftgrid3D object to be read from a file
fftgrid3D::fftgrid3D(char *file, double initialVal, int n1_inc, int n2_inc, int
    n3_inc, int n1_offset, int n2_offset, int n3_offset){
  if(!fftw_init_threads_called) {
```

```
      fftw_init_threads();
      fftw_plan_with_nthreads(2);
      fftw_init_threads_called++;
    }

    cout<<file<<endl;
    ifstream inFile(file, ios::in);
    inFile >> N1 >> N2 >> N3;

    //Increase the dimensions by n1_inc, n2_inc, n3_inc
    N1+=n1_inc;
    N2+=n2_inc;
    N3+=n3_inc;

    allocate(N1,N2,N3);
    grid_result = NULL;
    grid_result2 = NULL;
    plan_for = NULL;
    plan_rev = NULL;
    (*this)=initialVal;

    //Save the dimensions of the original array loaded from the file + offset
    N1_orig=N1-n1_inc+n1_offset;
    N2_orig=N2-n2_inc+n2_offset;
    N3_orig=N3-n3_inc+n3_offset;

    char temp;
    //Read in the grid from file handle
    for (int i=n1_offset; i<N1_orig; ++i)
      for (int j=n2_offset; j<N2_orig; ++j)
        for(int k=n3_offset; k<N3_orig; ++k) {
          inFile >> (*this)(i,j,k)[0] >> (*this)(i,j,k)[1]>>temp>>temp;
        }
    inFile.close();
}

//-----------------------------------------------------------------------
fftgrid3D::fftgrid3D(const fftgrid3D& tocopy){
  if(!fftw_init_threads_called) {
    fftw_init_threads();
    fftw_plan_with_nthreads(2);
    fftw_init_threads_called++;
  }
  N1=tocopy.N1;
  N2=tocopy.N2;
```

```
    N3=tocopy.N3;
    N1_orig=tocopy.N1_orig;
    N2_orig=tocopy.N2_orig;
    N3_orig=tocopy.N3_orig;
91  warned_range=0;
    warned_bndry=0;

    //Allocate space for the grid
    allocate(N1,N2,N3);
96  grid_result = NULL;
    grid_result2 = NULL;
    plan_for = NULL;
    plan_rev = NULL;
  }
101 //-------------------------------------------------------------------
  fftgrid3D::~fftgrid3D(void){
    fftw_free(grid);
    fftw_free(grid_result);
    fftw_free(grid_result2);
106 cleanup_fftw();
    grid=NULL;
    grid_result=NULL;
    grid_result2=NULL;
  }
111 //-------------------------------------------------------------------
  void fftgrid3D::initializeValueWithNoise(double val) {
    srand((unsigned)time(NULL));
    double offset;
    gridLoop{
116   offset = ((double)rand()/((double)RAND_MAX+1.0))*(2)-1;
      offset /= 1000;
      (*this)(i,j,k)[0] = val + offset;
      (*this)(i,j,k)[1]=0;
    }
121 }
  //-------------------------------------------------------------------
  void fftgrid3D::initializeNuclei(double p_inf, double p0_inf, double c_gamma,int r,
      fftgrid3D *phi) {
    //Note: This will only actually work for 2D grids (N3=1) right now
    //Will work on extending to 3D later
126 double c0 = 1 - c_gamma + p_inf*(2*c_gamma-1);
    //determine # of squares covered by a critical nucleus to get c_else
    int count = 0;
    for(int i=0;i<=2*r;++i) {
      for(int j=0;j<=2*r;++j) {
```

```
131         if((sq(i-r)+sq(j-r)) <= sq(r))
              ++count;
          }
        }
        //calculate the initial fraction of the grid that should be covered with nuclei
136     //calculate the number of nuclei needed to cover that amount
        int numnuclei = (int)lround(p0_inf*p_inf*N1*N2/count);
        //calculate the concentration the rest of the system should be at to get
        //overall concentration c0
        double c_else = (N1*N2*c0-numnuclei*count*c_gamma)/(N1*N2-numnuclei*count);
141     //set the system to that concentration
        (*this) = c_else;

        //Insert Nuclei so that none of them overlap and they can cross the borders
        //of the grid
146     int nfailed = 0, nplaced = 0;
        int rowstart, colstart, x0, y0, it, jt, occupied;
        //seed random number generator
        srand(time(NULL));
        while(nplaced < numnuclei) {
151       //rowstart and colstart represent the top left corner of the square that
          //encloses the circular nucleus
          rowstart = (int)((N1-1)*((double)rand()/((double)(RAND_MAX)+(double)(1))));
          colstart = (int)((N2-1)*((double)rand()/((double)(RAND_MAX)+(double)(1))));
          //(x0,y0) is the position of the center square of the nucleus
156       x0 = rowstart + r;
          y0 = colstart + r;

          //loop over the squares where the nucleus would be to make sure none are
          //already occupied
161       occupied = 0;
          for(int i=rowstart-1;i<=(rowstart+2*r+1);++i) {
            it = i; //use a fake index to deal with boundaries
            if(it<0)
              it+=N1;
166         if(it>=N1)
              it-=N1;
            for(int j=colstart-1;j<=(colstart+2*r+1);++j) {
              jt = j;
              if(jt<0)
171             jt+=N2;
              if(jt>=N2)
                jt-=N2;
              if((sq(i-x0) + sq(j-y0)) <= sq(r+1)) {
                if((*this)(it,jt,0)[0] == c_gamma) {
```

```
176                 occupied = 1;
                    break;
                  }
                }
              }
181           if(occupied == 1)
                break;
            }

            //if none of the squares were occupied then actually place the nucleus
186         if(occupied == 1) {
              ++nfailed;
              continue;
            }
            for(int i=rowstart;i<=rowstart+2*r;++i) {
191           it = i;
              if(it<0)
                it+=N1;
              if(it>=N1)
                it-=N1;
              for(int j=colstart;j<=colstart+2*r;++j) {
196             jt = j;
                if(jt<0)
                  jt+=N2;
                if(jt>=N2)
201               jt-=N2;
                if((sq(i-x0) + sq(j-y0)) <= sq(r)) {
                  (*this)(it,jt,0)[0] = c_gamma;
                  (*this)(it,jt,0)[1] = 0;
                  (*phi)(it,jt,0)[0] = 0.99;
206               (*phi)(it,jt,0)[1] = 0;
                }
              }
            }
            ++nplaced;
211       }
        }
        //-------------------------------------------------------------------
        /*
         * Uses the Hoshen-Kopelmann algorithm to count clusters and determine
216      * whether or not a phase has percolated. If (*this)(i,j,k) > threshold it is
         * counted as part of the percolating phase. Returns 1 if the system has
         * percolated, 0 if it has not.
        */
        int fftgrid3D::hoshenKopelman(double threshold) {
```

```
221   int maxsize = N1*N2;
      int *badlabels=new int[maxsize/2];
      for(int p = 0; p<maxsize/2; ++p)
        badlabels[p]=p;
      int *temp;
226   temp = new int[maxsize]; //store in row-major order
      int up,left,curlabel=1;
      for(int i=0; i < maxsize; ++i) {
        if(grid[i][0] > threshold) {
          if(i < N2) //top row in row major order
231         up = 0;
          else
            up = temp[i-N2];

          if(!(i%N2)) //left column
236         left = 0;
          else
            left = temp[i-1];

          switch(!!up + !!left) {
241         case 0: //new cluster
              temp[i] = curlabel;
              curlabel = curlabel+1;
              break;
            case 1: //connected either to the left or above
246           temp[i] = max(up,left);
              break;
            case 2: //connected both to the left and above
              //need to fix this
              int q=up,r=up,s;
251           while(badlabels[q] != q)
                q = badlabels[q];
              while(badlabels[r] != r) {
                s = badlabels[r];
                badlabels[r] = q;
256             r = s;
              }
              int xl = q;

              q=left;
261           r=left;
              while(badlabels[q] != q)
                q = badlabels[q];
              while(badlabels[r] != r) {
                s = badlabels[r];
```

```
266             badlabels[r] = q;
                r = s;
              }
              int yl = q;

271           badlabels[xl] = yl;
              temp[i] = yl;
              break;
            }
          } else {
276         temp[i] = 0;
          }
        }

        //now replace any bad labels
281     int n;
        for(int i=0;i<maxsize;++i) {
            if(temp[i] == 0)
              continue;
            else {
286           n=badlabels[temp[i]];
              while(n!=temp[i]) {
                temp[i] = n;
                n = badlabels[n];
              }
291         }
        }

        //determine if there is a cluster percolating from top to bottom
        int isperc = 0, inflabel = -1;
296     for(int i=0; i<N2;++i) {
          for(int j=0;j<N2;++j) {
            if((temp[i] == temp[N2*(N1-1)+j]) && temp[i] != 0) {
              inflabel = temp[i];
              break;
301         }
          }
          if(inflabel!=-1)
            break;
        }
306     //see if it all reaches left to right
        int right = 0;
        left = 0;
        for(int i=1;i<N1-1;++i) {
          if(temp[N2*i] == inflabel)
```

```
311         left = 1;
          if(temp[N2*i+N2-1] == inflabel)
            right = 1;
        }
        if(left == 1 && right == 1)
316       isperc = 1;

        delete[] badlabels;
        delete[] temp;
        badlabels = NULL;
321     temp == NULL;
        return isperc;
      }
      //-------------------------------------------------------------------
      double fftgrid3D::volumeFraction(double threshold) {
326     int num=0, max=N1*N2*N3;
        for(int i=0; i<max; ++i) {
          if(grid[i][0] >= threshold)
            ++num;
        }
331     return (double)num/((double)max);
      }
      //-------------------------------------------------------------------
      void fftgrid3D::writeToFileComplex(char *file){
        //For 2D write entire row on one line
336     //3D writes entire plane on one line
        //i.e. line breaks only when row counter is updated
        //Comma separated
        ofstream outFile(file);
        outFile << N1 << "," << N2 << "," << N3 << endl;
341
        for (int i=0; i<N1; i++){
          for (int j=0; j<N2; j++){
            for (int k=0; k<N3; k++) {
              //TODO: Have this work properly when imaginary part = 0
346           outFile << (*this)(i,j,k)[0];
              if((*this)(i,j,k)[1] > 0)
                outFile<<"+";
              outFile << (*this)(i,j,k)[1] << "i";
              if((N2-1) != j)
351             outFile<<",";
            }
          }
          outFile << endl;
        }
```

```
356     outFile.close();
    }
    //-----------------------------------------------------------------------
    void fftgrid3D::writeToFile(char *file){
      //For 2D write entire row on one line
361   //3D writes entire plane on one line
      //i.e. line breaks only when row counter is updated
      //Comma separated
      ofstream outFile(file);
      outFile << N1 << " " << N2 << " " << N3 << endl;
366
      for (int i=0; i<N1; i++){
        for (int j=0; j<N2; j++){
          for (int k=0; k<N3; k++) {
            outFile << (*this)(i,j,k)[0]<<' ';
371       }
        }
        outFile << endl;
      }
      outFile.close();
376 }
    //-----------------------------------------------------------------------
    void fftgrid3D::allocate(int N1, int N2, int N3){
      grid = (fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N1*N2*N3);
    }
381 //-----------------------------------------------------------------------
    void fftgrid3D::naturallog() {
      double mod,arg,max=N1*N2*N3;
      for(int i=0; i<max; ++i) {
    #ifdef __SSE3__
386     sse3ComplexMagnitude(grid[i], &mod);
    #else
        mod = sqrt(sq(grid[i][0]) + sq(grid[i][1]));
    #endif
        arg = atan(grid[i][1]/grid[i][0]);
391     grid[i][0] = log(mod);
        grid[i][1] = arg;
      }
    }
    //-----------------------------------------------------------------------
396 void fftgrid3D::init_fftw() {
      if(1 == N3) {
        grid_result = (double*)fftw_malloc(sizeof(double)*N1*N2);
        grid_result2 = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*N1*(N2/2+1));
        plan_for = fftw_plan_dft_r2c_2d(N1,N2,grid_result,grid_result2,FFTW_MEASURE);
```

```
401       plan_rev = fftw_plan_dft_c2r_2d(N1,N2,grid_result2 ,grid_result ,FFTW_MEASURE);
        } else {
          grid_result = (double*)fftw_malloc(sizeof(double)*N1*N2*N3);
          grid_result2 = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*N1*N2*(N3/2+1));
          plan_for = fftw_plan_dft_r2c_3d(N1,N2,N3,grid_result ,grid_result2 ,FFTW_MEASURE);
406       plan_rev = fftw_plan_dft_c2r_3d(N1,N2,N3,grid_result2 ,grid_result ,FFTW_MEASURE);
        }
      }
      //-------------------------------------------------------------------------
      void fftgrid3D::cleanup_fftw() {
411     fftw_destroy_plan(plan_for);
        fftw_destroy_plan(plan_rev);
      }
      //-------------------------------------------------------------------------
      fftgrid3D fftgrid3D::fft() {
416     //Allocate grid_result iff fft() is actually called, and only once
        //add padding to do in place transform
        //Only create plans if we are actual transforming this instance
        // and only create plans once per instance
        if(!plan_for || !grid_result) //redundant since both are inited in same place
421       init_fftw();
        fftgrid3D answer(N1,N2,N3);
        //populate grid_result
        int max = N1*N2*N3;
        for(int i=0; i<max; ++i) {
426       grid_result[i] = grid[i][0];
        }
        fftw_execute(plan_for);
        //Store computed transform values in answer
        int jmax,kmax,index;
431     if(1==N3) {
          jmax = (N2/2+1);
          kmax = N3;
        } else {
          jmax = N2;
436       kmax = (N3/2+1);
        }
        max = N1*jmax*kmax;
        fftw_complex *answergrid = answer.getGrid();
        int i=0,j=0;
441     for(int a=0;a<max;++a) {
          index = j+N2*i;
          answergrid[index][0] = grid_result2[a][0];
          answergrid[index][1] = grid_result2[a][1];
          ++j;
```

```
446       if(j>=jmax) {
            ++i; j=0;
          }
        }
        return answer;
451   }
      //-----------------------------------------------------------------------
      fftgrid3D fftgrid3D::ifft() {
        //Allocate grid_result iff fft() is actually called, and only once
        //Only create plans if we are actually transforming this instance
456     // and only create plans once per instance
        if(!plan_rev || !grid_result)
          init_fftw();
        fftgrid3D answer(N1,N2,N3);
        int i=0,j=0,max,jmax,kmax,index;
461     if(1==N3) {
          jmax = (N2/2+1);
          kmax = N3;
        } else {
          jmax = N2;
466       kmax = (N3/2+1);
        }
        max = N1*jmax*kmax;
        for(int a=0;a<max;++a) {
          index = (j+N2*i);
471       grid_result2[a][0] = grid[index][0];
          grid_result2[a][1] = grid[index][1];
          ++j;
          if(j>=jmax) {
            ++i; j=0;
476       }
        }
        fftw_execute(plan_rev);
        max = N1*N2*N3;
        fftw_complex *answergrid = answer.getGrid();
481     for(int i=0;i<max;++i)
          answergrid[i][0] = grid_result[i];
        answer = answer/max;//normalize the inverse transform
        return answer;
      }
```

A.3 sse2.h

```
#ifndef SSE2_H
#define SSE2_H
#ifdef __SSE2__

#include <emmintrin.h>
//fftw_malloc automatically aligns data properly so easy to use SSE operations

extern inline void sseAdd(double *lhs, double *rhs, double *result) {
  register __m128d l, r, res;
  l = _mm_load_pd(lhs);
  r = _mm_load_pd(rhs);
  res = _mm_add_pd(l,r);
  _mm_store_pd(result,res);
}

extern inline void sseSub(double *lhs, double *rhs, double *result) {
  register __m128d l, r, res;
  l = _mm_load_pd(lhs);
  r = _mm_load_pd(rhs);
  res = _mm_sub_pd(l,r);
  _mm_store_pd(result,res);
}

extern inline void sseComplexConstMult(double *lhs,double rhs,double *result) {
  register __m128d l, r, res;
  l = _mm_load_pd(lhs);
  r = _mm_set1_pd(rhs);
  res = _mm_mul_pd(l,r);
  _mm_store_pd(result,res);
}

extern inline void sseComplexMult(double *lhs, double *rhs, double *result) {
  register __m128d l, r, temp1, temp2, swapr, res;
  l = _mm_load_pd(lhs);//l0,l1
  r = _mm_load_pd(rhs);//r0,r1
  swapr = _mm_shuffle_pd(r,r,_MM_SHUFFLE2(0,1));//r1,r0
  temp1 = _mm_mul_pd(l,r);//l0*r0,l1*r1
  temp2 = _mm_mul_pd(l,swapr);//l0*r1,l1*r0
  l = _mm_shuffle_pd(temp1, temp2, _MM_SHUFFLE2(0,0));//l0*r0,l0*r1
  r = _mm_shuffle_pd(temp1, temp2, _MM_SHUFFLE2(1,1));//l1*r1, l1*r0
  res = _mm_sub_pd(l,r);
  _mm_store_pd(result,res);
}
```

```
45  extern inline void sseComplexConstDiv(double *lhs,double rhs,double *result) {
      register __m128d l,r,res;
      l = _mm_load_pd(lhs);
      r = _mm_set1_pd(rhs);
      res = _mm_div_pd(l,r);
50    _mm_store_pd(result,res);
    }

    extern inline void sseComplexDiv(double *lhs, double *rhs, double *result) {
      register __m128d l, r, denom, temp1, swap1, temp3, res;
55    l = _mm_load_pd(lhs);
      swap1 = _mm_shuffle_pd(l,l, _MM_SHUFFLE2(0,1));
      r = _mm_load_pd(rhs);
      denom = _mm_mul_pd(r,r); //c*c, d*d
      temp1 = _mm_mul_pd(l,r); //a*c, b*d
60    temp3 = _mm_mul_pd(swap1,r); //b*c, a*d
      l = _mm_shuffle_pd(denom, temp1, _MM_SHUFFLE2(0,0)); //c*c, a*c
      r = _mm_shuffle_pd(denom, temp1, _MM_SHUFFLE2(1,1)); //d*d, b*d
      denom = _mm_add_pd(l,r); //c*c+d*d, a*c+b*d
      temp1 = _mm_shuffle_pd(denom, denom, _MM_SHUFFLE2(1,1));//a*c+b*d,a*c+b*d
65    denom = _mm_shuffle_pd(denom, denom, _MM_SHUFFLE2(0,0));//c*c+d*d,c*c+d*d
      swap1 = _mm_shuffle_pd(temp3,temp3, _MM_SHUFFLE2(0,0));//b*c,b*c
      temp3 = _mm_shuffle_pd(temp3,temp3, _MM_SHUFFLE2(1,1));//a*d, a*d
      l = _mm_sub_pd(swap1,temp3);//b*c-a*d,b*c-a*d
      temp1 = _mm_shuffle_pd(temp1, l, _MM_SHUFFLE2(1,0));//a*c+b*d,b*c-a*d
70    res = _mm_div_pd(temp1,denom);
      _mm_store_pd(result,res);
    }
    #endif //__SSE2__
    #endif //SSE2_H
```

A.4 sse3.h

```
#ifndef SSE3_H
#define SSE3_H
#ifdef __SSE3__
#include <pmmintrin.h>

extern inline void sse3ComplexMult(double *lhs, double *rhs, double *result) {
  register __m128d l, r, swapr, temp1, temp2, res;
  l = _mm_load_pd(lhs);//l0,l1
  r = _mm_load_pd(rhs);//r0,r1
  swapr = _mm_shuffle_pd(r,r,_MM_SHUFFLE2(0,1));//r1,r0
  temp1 = _mm_mul_pd(l,r);//l0*r0,l1*r1
  temp2 = _mm_mul_pd(l,swapr);//l0*r1,l1*r0
  res = _mm_hsub_pd(temp1,temp2);//l0*r0-l1*r1,l0*r1-l1*r0
  _mm_store_pd(result,res);
}

extern inline void sse3ComplexDiv(double *lhs, double *rhs, double *result) {
  register __m128d l,r,swapl,temp1,temp2,denom,res;
  l = _mm_load_pd(lhs);//a,b
  swapl = _mm_shuffle_pd(l,l,_MM_SHUFFLE2(0,1));//b,a
  r = _mm_load_pd(rhs);//c,d
  temp1 = _mm_mul_pd(l,r);//a*c, b*d
  temp2 = _mm_mul_pd(swapl,r);//b*c,a*d
  l = _mm_mul_pd(r,r);//c*c,d*d
  denom = _mm_hadd_pd(l,l);//c*c+d*d,c*c+d*d
  l = _mm_shuffle_pd(temp1,temp2,_MM_SHUFFLE2(0,0));//a*c,b*c
  r = _mm_shuffle_pd(temp1,temp2,_MM_SHUFFLE2(1,1));//b*d,a*d
  temp1 = _mm_addsub_pd(l,r);//a*c+b*d,b*c-a*d
  res = _mm_div_pd(temp1,denom);
  _mm_store_pd(result,res);
}

extern inline void sse3ComplexMagnitude(double *val, double *result) {
  register __m128d v, temp, res;
  v = _mm_load_pd(val);//v0,v1
  temp = v;
  v = _mm_mul_pd(v,temp);//v0*v0,v1*v1
  res = _mm_hadd_pd(v,v);//v0*v0+v1*v1,v0*v0+v1*v1
  res = _mm_sqrt_pd(res);//get the sqrt
  _mm_store_sd(result,res);
}
#endif
#endif
```

A.5 phasefield3D.h

```
#ifndef _PHASEFIELD3D_H_
#define _PHASEFIELD3D_H_

#include <iostream>
#include <complex>
#include "grid3D.h"
#include "fftgrid3D.h"
#include "macros3D.h"
using namespace std;
//Function definitions
//-------------------------------------------------------------------------
double binary_alloy3D_fft(fftgrid3D*,fftgrid3D*,double,int,int,double,char*,bool);
double binary_alloy3D_fft_chonly(fftgrid3D*,double,int,int,double,char*,bool);
#endif
```

A.6 phasefield3D.cpp

```
#include "phasefield3D.h"

//This function solves coupled cahn-hilliard and allen-cahn to simulate a
//binary system. Uses a semi-implicit method and FFT instead of Forward Euler
//-------------------------------------------------------------------------
double binary_alloy3D_fft(fftgrid3D* phi, fftgrid3D* c,double h, int iterations, int
    outputevery,double p_inf, char *dirname, bool stoponperc) {
  double result = -1.0;
  char outfile[128];
  //define constants (same as values I have been using in MATLAB)
  outputevery=50;
  iterations=40000;
  double dt = 0.001;
  double m_phi = 1.;
  double w_phi = 10.;
  double m_c = 0.0001;
  double eps_phi = 0.005;
  double eps_c = 0.005;
  double L = 20.;
  double T0 = 0.4;
  double tau = 1.1;

  //define Laplacian eigenvalue matrix
  //***Only set up for 2D right now
  fftgrid3D Leig(*phi);
  double kx,ky;
  int Nx = Leig.getDimension(X_DIM),Ny=Leig.getDimension(Y_DIM);
  int Nz = Leig.getDimension(Z_DIM);
  gridLoop3D(Leig) {
    kx = 2.0*PI*(double)i/(double)Nx;
    ky = 2.0*PI*(double)j/(double)Ny;
    Leig(i,j,k)[0] = 2./3.*(cos(kx)*cos(ky)-1) + 4./3.*(cos(kx)+cos(ky)-2);
    Leig(i,j,k)[1] = 0.;
  }

  //define some dummy variables to make later lines shorter
  double dum1 = m_phi*sq(eps_phi)*dt/sq(h);
  double dum2 = m_phi*w_phi*dt;
  double dum3 = L*T0*(1.-tau)*dt;
  double dum4 = m_c*sq(eps_c)*dt/(sq(h)*sq(h));
  fftgrid3D dum5 = Leig*(m_c*T0*dt/sq(h));
  fftgrid3D dum6 = (m_c*dum3/sq(h))*Leig;
  dum6(0,0,0)[0] = 0.0; //ugly hack that might work
```

```
    dum6(0,0,0)[1] = 0.0;

    //define LHS terms of solved allen-cahn and cahn-hilliard
46  fftgrid3D phi_lhs = 1.0 - dum1*Leig;
    fftgrid3D c_lhs = 1.0 + dum4*sq(Leig);

    //Perform initial transform
    fftgrid3D phi2(*phi),phi3(*phi),phi4(*phi),phi5(*phi),c2(*c);
51  fftgrid3D phi_hat(*phi),phi2_hat(phi2),phi3_hat(phi3),phi4_hat(phi4);
    fftgrid3D phi5_hat(phi5),c_hat(*c),c2_hat(c2);
    phi_hat = phi->fft();
    c_hat = c->fft();
    //Begin iterating
56  for (int n=0; n<iterations+1; n++){
      //Write output, if necessary
      if (!(n%outputevery)){
        sprintf(outfile,"%s/step_%6.6i.phi",dirname,n);
        cout << "writing output: " << outfile << endl;
61      cout << phi->volumeFraction(0.5) << endl;
        phi->writeToFile(outfile);

        sprintf(outfile,"%s/step_%6.6i.c",dirname,n);
        c->writeToFile(outfile);
66    }
      phi2 = sq(*phi);
      phi3 = (*phi)*phi2;
      phi4 = (*c)*(*phi) - (*c)*phi2;
      phi5 = (*phi) - phi2;
71    c2 = (*c) / (1.0 - (*c));
      c2.naturallog();
      phi2_hat = phi2.fft();
      phi3_hat = phi3.fft();
      phi4_hat = phi4.fft();
76    phi5_hat = phi5.fft();
      c2_hat = c2.fft();

      //Calculate the new phi_hat, c_hat
      phi_hat = -dum2/2.0*phi_hat + 3.0*dum2/2.0*phi2_hat - dum2*phi3_hat
81      -12.0*m_phi*dum3*phi4_hat + 6.0*m_phi*dum3*phi5_hat + phi_hat;

      c_hat = dum5*c2_hat + 6.0*dum6*phi2_hat - 4.0*dum6*phi3_hat + c_hat;

      c_hat = c_hat/c_lhs;
86    (*c) = c_hat.ifft();
```

```
      phi_hat = phi_hat/phi_lhs;
      (*phi) = phi_hat.ifft();

91    if(stoponperc && phi->hoshenKopelman(0.5)) {
        //System has percolated, set perctime and break
        result = dt*n;
        break;
      }
96    if(phi->volumeFraction(0.5) >= 0.99*p_inf) {
        //Not going to percolate, break
        break;
      }
    }
101 //End iterating
    //Save final system state
    sprintf(outfile,"%s/final_step.phi",dirname);
    phi->writeToFile(outfile);
    sprintf(outfile,"%s/final_step.c",dirname);
106 c->writeToFile(outfile);
    return result;
  }

  //-------------------------------------------------------------------------
111 double binary_alloy3D_fft_chonly(fftgrid3D* c,double h, int iterations, int
      outputevery,double p_inf, char *dirname, bool stoponperc) {
    //debug line
    double result = -1.0;
    char outfile[128];
    //define constants (same as values I have been using in MATLAB)
116 outputevery=50;
    iterations=20000;
    double dt = 0.00001;
    double m_c = 0.001;
    double eps_c = 0.015;
121 double f_hom_max = 1;
    double c_beta = 0.880797077978;
    double beta_threshold = 0.85*c_beta;
    double c_alpha = 1-c_beta;

126 //define Laplacian eigenvalue matrix
    //***Only set up for 2D right now
    fftgrid3D Leig(*c);
    double kx,ky;
    int Nx = Leig.getDimension(X_DIM),Ny=Leig.getDimension(Y_DIM);
131 int Nz = Leig.getDimension(Z_DIM);
```

```
    gridLoop3D(Leig) {
      kx = 2.0*PI*(double)i/(double)Nx;
      ky = 2.0*PI*(double)j/(double)Ny;
      Leig(i,j,k)[0] = 2./3.*(cos(kx)*cos(ky)-1) + 4./3.*(cos(kx)+cos(ky)-2);
136   Leig(i,j,k)[1] = 0.;
    }

    //define some dummy variables to make later lines shorter
    double dum1 = 32*f_hom_max/sq(sq(c_beta-c_alpha));
141 fftgrid3D dum2 = 2/sq(h)*Leig;
    fftgrid3D dum3 = 3*(c_alpha+c_beta)/sq(h)*Leig;
    fftgrid3D dum4 = (sq(c_alpha)+4*c_alpha*c_beta+sq(c_beta))/sq(h)*Leig;
    double dum5 = m_c*sq(eps_c)*dt/sq(sq(h));

146 //define LHS terms of solved allen-cahn and cahn-hilliard
    fftgrid3D c_lhs = 1.0 + dum5*sq(Leig);

    //Perform initial transform
    fftgrid3D c2(*c),c3(*c);
151 fftgrid3D c_hat(*c),c2_hat(c2),c3_hat(c3);
    c_hat = c->fft();
    //Begin iterating
    for (int n=0; n<iterations+1; n++){
      //Write output, if necessary
156   if (!(n%outputevery)){
        sprintf(outfile,"%s/step_%6.6i.phi",dirname,n);
        cout << "writing output: " << outfile << endl;

        cout << c->volumeFraction(beta_threshold) << endl;
161     sprintf(outfile,"%s/step_%6.6i.c",dirname,n);
        c->writeToFile(outfile);
      }
      c2 = sq(*c);
      c3 = (*c)*c2;
166   c2_hat = c2.fft();
      c3_hat = c3.fft();

      //Calculate the new c_hat
      c_hat = c_hat + m_c*dum1*dt*(dum2*c3_hat - dum3*c2_hat + dum4*c_hat);
171   c_hat = c_hat/c_lhs;
      (*c) = c_hat.ifft();

      if(stoponperc && c->hoshenKopelman(beta_threshold)) {
        //System has percolated, set perctime and break
176     result = dt*n;
```

```
        break;
      }
      if(c->volumeFraction(beta_threshold) >= 0.99*p_inf) {
        //Not going to percolate, break
181     break;
      }
    }
    //End iterating
    //Save final system state
186 sprintf(outfile,"%s/final_step.c",dirname);
    c->writeToFile(outfile);
    return result;
  }
```

A.7 main.cpp

```
#include <iostream>
#include <unistd.h>
#include "grid3D.h"
#include "phasefield3D.h"
using namespace std;

//------------------------------------------------------------------------
int main(int argc, char *argv[]){
  //Use constant h so that the system is actually larger for larger N
  double h=1./256.;
  int iterations=40000;
  int outputEvery=50;
  //create variables with default values, parse cmd line options
  //for actual values to use
  int N = 32;
  double p_inf = 0.6;
  double p0_inf = 0.1;
  char *dirname = "output";
  char* filename = new char[128];
  int spinodal = 0;
  //grid3D* initial_condition, *initial_phi, *initial_c;
  fftgrid3D *initial_phi_fft=NULL, *initial_c_fft=NULL;
  int option_char;

  // Handle command line options
  bool inputFileSupplied=false;
  while ((option_char = getopt(argc, argv, "f:N:d:p:i:s:h")) != -1)
    switch (option_char){
    case 'f': {
      inputFileSupplied=true;
      filename=optarg;
      char *cfile, *phifile;
      cfile = new char[strlen(filename)+2];
      phifile = new char[strlen(filename)+4];
      strcpy(cfile,filename);
      strcpy(phifile,filename);
      strcat(cfile,".c");
      strcat(phifile,".phi");
      initial_c_fft = new fftgrid3D(cfile);
      initial_phi_fft = new fftgrid3D(phifile);
      delete cfile;
      delete phifile;
      break;
```

```
        }
      case 'N':
46      N = atoi(optarg);
        break;
      case 'd':
        dirname = optarg;
        break;
51    case 'p':
        p_inf = atof(optarg);
        break;
      case 'i':
        p0_inf = atof(optarg);
56      break;
      case 's':
        spinodal = atoi(optarg);
        break;
      case 'h':
61      cout << "-f filename \n\
    \t\tinitial_phi loaded from filename.phi, initial_c from filename.c\n\
    -N \n\
    \t\tinitial system size\n\
    -d\n\
66  \t\tdirectory to save output into\n\
    -p\n\
    \t\tvalue for p_inf\n\
    -i\n\
    \t\tvalue for p0_inf\n\
71  -s\n\
    \t\tRun the spinodal decomposition (C-H only) version.\n\
    -h\n\
    \t\tDisplay this help message.\n";
        exit(0);
76      break;
      }

    if (!inputFileSupplied){
      cout << "No input file supplied!" << endl;
81    if(!spinodal) {
        //L= 20, tau=1.1
        //c_gamma=exp(-L(1-tau))/(1+exp(-L*(1-tau)))
        double c_gamma = 0.880797077978;
        int radius = 2;
86      initial_phi_fft = new fftgrid3D(N,N,1);
        initial_c_fft = new fftgrid3D(N,N,1);
        *initial_phi_fft = 0.01;
```

```
        initial_c_fft->initializeNuclei(p_inf,p0_inf,c_gamma,radius,initial_phi_fft);
      } else {
91      double c_beta=0.880797077978;
        double c_alpha=1.0-c_beta;
        double c_0 = c_alpha+p_inf*(c_beta-c_alpha);
        initial_c_fft = new fftgrid3D(N,N,1);
        initial_c_fft->initializeValueWithNoise(c_0);
96    }
    }

    double perctime = -1.0;
    if(!spinodal)
101   perctime = binary_alloy3D_fft(initial_phi_fft,initial_c_fft,h,iterations,
          outputEvery,p_inf,dirname,false);
    else
      perctime = binary_alloy3D_fft_chonly(initial_c_fft,h,iterations,outputEvery,p_inf
          ,dirname,false);
    cout<<perctime<<endl;
    delete filename;
106 if(initial_phi_fft)
      delete initial_phi_fft;
    if(initial_c_fft)
      delete initial_c_fft;
    initial_phi_fft = NULL;
111 initial_c_fft = NULL;
    return 0;
  }
```

Bibliography

[1] G. B. McFadden A. A. Wheeler and W. J. Boettinger. Phase-field model for solidification of a eutectic alloy. *Proceedings: Mathematical, Physical and Engineering Sciences*, 452(1946):495–525, 1996.

[2] I. Orgzall B. Lorenz and H.-O. Heuer. Universality and cluster structures in continuum models of percolation with two different radius distributions. *Journal of Physics A: Mathematical and General*, 26:4711–4722, 1993.

[3] Sergey Fomel and Jon F. Claerbout. Exploring three-dimensional implicit wavefield extrapolation with the helix transform. *Stanford Exploration Project*, 95:43–60.

[4] Steven W. Haan and Robert Zwanzig. Series expansions ina continuum percolation problem. *Journal of Physics A: Mathematical and General*, 10:1547–1555, 1977.

[5] J. E. Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461.

[6] Mohan K. Phani and Deepak Dhar. Continuum percolation with discs having a distribution of radii. *Journal of Physics A: Mathematical and General*, 17:L645–L649, 1984.

[7] G. E. Pike and C. H. Seager. Percolation and conductivity: A computer study. i. *Physical Review B*, 10(4):1421–1434, 1974.

[8] M. D. Rintoul and S. Torquato. Precise determination of the critical threshold and exponents in a three-dimensional continuum percolation model. *Journal of Physics A: Mathematical and General*, 30:585–592, 1997.

[9] Samuel M. Allen Robert W. Balluffi and W. Craig Carter. *Kinetics of Materials*, chapter 18. John Wiley & Sons, Inc., 2005.

[10] C. H. Seager and G. E. Pike. Percolation and conductivity: A computer study. ii. *Physical Review B*, 10(4):1435–1446, 1974.

[11] Dietrich Stauffer and Amnon Aharony. *Introduction to Percolation Theory*, volume 2. Taylor & Francis, 1992.

[12] Ronal L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*, chapter 30, pages 822–848. MIT Press and McGraw-Hill, 2001.

[13] A. R. Kerstein W. T. Elam and J. J. Rehr. Critical properties of the void percolation problem for spheres. *Physical Review Letters*, 52(17):1516–1519, 1984.

[14] C. Beckermann W.J. Boettinger, J.A. Warren and A. Karma. Phase field simulation of solidification. *Annual Review of Materials Research*, 32:163–194, 2002.