DEVELOPMENT OF A FAST AND ACCURATE TIME STEPPING SCHEME FOR THE FUNCTIONALIZED CAHN-HILLIARD EQUATION AND APPLICATION TO A GRAPHICS PROCESSING UNIT

By

Jaylan Stuart Jones

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Applied Mathematics and Physics - Doctor of Philosophy

2013

ABSTRACT

DEVELOPMENT OF A FAST AND ACCURATE TIME STEPPING SCHEME FOR THE FUNCTIONALIZED CAHN-HILLIARD EQUATION AND APPLICATION TO A GRAPHICS PROCESSING UNIT

By

Jaylan Stuart Jones

This dissertation explores and develops time-stepping schemes for computing solutions to the Functionalized Cahn-Hilliard (FCH) model. It is important to find a scheme that is both fast enough to compute evolution to the long-time states and to give enough accuracy to capture important geometric events. The FCH model is relatively new, and very little work has been done to develop efficient numerical schemes for its simulation, so much of this work is based on the extensive work done on the Cahn-Hilliard (CH) model. For each of the methods, the spatial approximation is computed with a Fourier spectral method. All of the schemes are adapted to be computed on a graphics processing unit (GPU) which gives significant improvements in the speed of the simulation.

First, an implicit-explicit (IMEX) method will be introduced that is based on a convex splitting of the right hand side of the equation. This splitting guarantees that the solutions will decay in energy for any size time step, which gives numerical stability for very large time steps. With this splitting, a novel iterative method for solving the implicit portion greatly improves the numerical efficiency.

Second is the development of a fully implicit method that attains high accuracy. The method uses a conjugate gradient method to solve the Netwon's method iterations, and is preconditioned using a physics based approximation to the operator that is easy to invert and numerically efficient.

Lastly, exponential time differencing (ETD) methods are derived for the Cahn-Hilliard and Functionalized Cahn-Hilliard Equations. The ETD methods are all explicit which affords computation speed, and higher order versions are natural extensions giving accurate time stepping.

Finally, numerical experiments for the three types of methods compare the accuracy and speed. These simulations are performed for both fixed time-step simulations as well as adaptive time steps. This gives a clear picture of the strengths and weaknesses, and it gives enough information to determine which time-stepping method will work best for approximating solutions to the FCH equation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Order Parameters and Phase Field Models

In 1937, Lev Landau published two papers on the nature of phase transitions [1, 2], wherein he used the idea of an order parameter to describe the nature of the material at different points in space. This parameter was a function of space and could be used, for example, to understand the average direction of spin at each point in a magnetic material. The order parameter was later used in the Ginzburg-Landau theory of superconductivity [3]. This order parameter function was the foundation for phase field models many years later.

The phase field model was introduced in 1983 by George J. Fix to model first order liquid to solid phase transitions [4]. Later, J.S. Langer compared the model to similar models in solidification theory and described the value of such a model when describing the physics of phase transitions [5]. Since then, phase field models have been used to describe the physics of many different systems including grain growth and coarsening, microstructure evolution in thin films, surface-stress-induced pattern formation, crack propagation, crystal growth in the presence of strain, multiphase fluid flow, stress and electromagnetic driven void migration, tumor growth, vesicle dynamics, and multicomponent interdiffusion (see [6] and [7] for a list of various references). In general, phase field models have become a valuable tool in analyzing and modeling systems where multiple phases of a material separate due to high interaction energies.

Phase field models are powerful in describing microstructural evolution because they eliminate the need to track the evolving fronts that describe phase boundaries. In one dimension, an interface-tracking model can easily describe the sharp interfaces and calculate their motions, but when describing complex phase separations in two or three dimensions, it is no longer practical to track the interface directly. Instead, a phase function, $u(t, \vec{x})$ is introduced into the model to delineate the amount of each specific phase that exists at that place and time. The evolution of the phase function is governed by a set of equations which balance diffusion against driving forces. The primary advantage of a phase function model is that the boundaries between phases have finite thickness and are therefore easier to analyze mathematically. Phase field formulations can also simplify the numerical simulations of such systems because the derivatives of the phase function remain finite across the interfaces.

Phase field models are typically used to track the motion of interfaces, and the thermodynamic and kinetic coefficients are chosen to match the coefficients in a corresponding sharp-interface model. The phase function evolves in time, and when a sharp-interface representation is desired, the solution of the phase function can be projected into the sharp-interface model. This research will be working with a phase field model similar in nature to the well known Cahn-Hilliard model.

1.2 The Cahn-Hilliard Model

What is now commonly termed the Cahn-Hilliard model was originally developed by van der Waals in 1893 [8, 9]. The model was essentially forgotten until 1958 when, without knowledge of van der Waals' work, John Cahn and John Hilliard rederived the model to describe the separation of liquid metal alloys as they coarsened due to cooling [10]. The

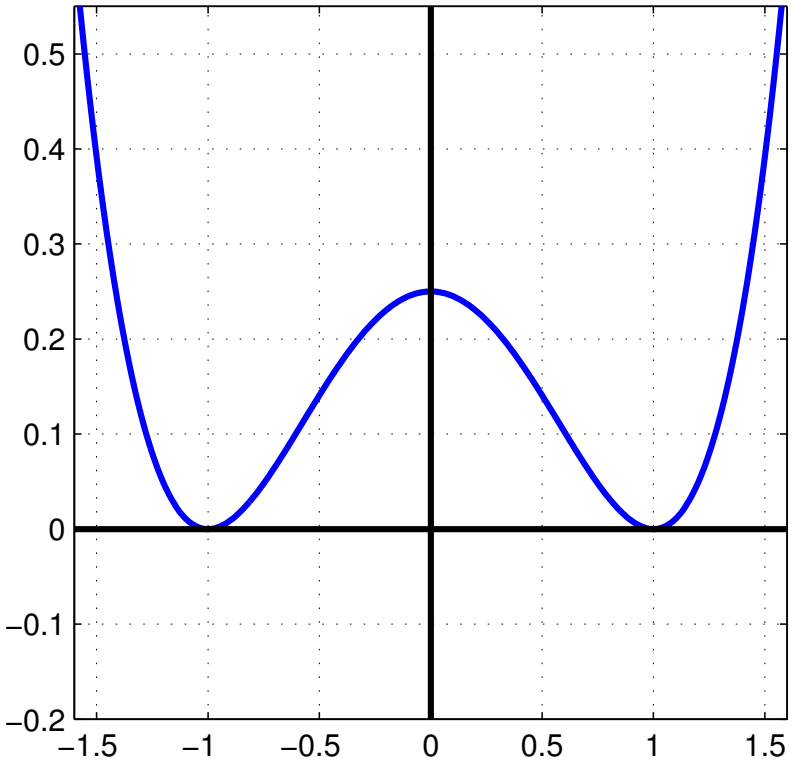


Figure 1.1 Potential energy function $W(u)$. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

model is expressed in the energy form

$$\mathcal{E}_{CH}(u) = \int_{\Omega} \frac{\varepsilon^2}{2} |\nabla u|^2 + W(u)\, dx, \tag{1.1}$$

where ∇u is the density gradient of one of the phases and $W(u)$ is a function that represents the potential energy of the mixed states of the phases. A symmetric, double-well potential is typically used for $W(u)$, which defines local energy minima for pure states with $u = \pm 1$ and an endothermic energy for the mixed states (Figure 1.1). In 1989, Pego defined the chemical potential [11]

$$\mu = \frac{\delta \mathcal{E}_{CH}}{\delta u}.$$

We calculate the variational derivative $\frac{\delta \mathcal{E}_{CH}}{\delta u}$ using a formula similar to the Euler-Lagrange equation. If

$$\mathcal{F}\left[\rho\left(\vec{r}\right)\right] = \int f\left(\vec{r}, \rho\left(\vec{r}\right), \nabla\rho\left(\vec{r}\right)\right) d\vec{r}$$

then

$$\mu = \frac{\delta \mathcal{F}\left[\rho\right]}{\delta \rho} = \frac{\partial f}{\partial \rho} - \nabla \cdot \frac{\partial f}{\partial\left(\nabla\rho\right)}. \tag{1.2}$$

Applying formula (1.2) to (1.1) and substituting into Pego's equation for the chemical potential we obtain

$$\mu = -\varepsilon^2 \Delta u + W'\left(u\right),$$

so that the diffusion equation becomes $u_t = \Delta\mu = \Delta\left(-\varepsilon^2\Delta u + W'\left(u\right)\right)$ from Fick's second law of diffusion [12]. When written this way, it appears in the form of a conservation law $u_t = \nabla \cdot j\left(x\right)$ where $j\left(x\right) = \nabla\mu$. The conserved quantity is the total mass of each of the phases in the system. This makes the evolution of the Cahn-Hilliard model a mass conserving H^{-1} gradient flow for the energy (1.1). The resulting differential equation is commonly called the Cahn-Hilliard equation.

$$u_t = -\Delta\left(\varepsilon^2\Delta u - W'\left(u\right)\right). \tag{1.3}$$

Due to the importance of the H^{-1} gradient flow and inner product to this work, it is useful to recall some definitions. First, L^p is the Lebesgue function space defined such that a function f is in L^p if $\int |f|^p \, dx < \infty$ on the domain of interest (which could be unbounded). Further H^s, for non-negative integers s, is the Sobolev space of functions that contains all functions f such that f is in L^2 and all of its first s weak derivatives are in L^2. H^{-1} is not a Sobolev space, nor is it even a space of functions, but rather the space of distributions (an

extension of functions) that acts as a dual space to H^1.

We introduce the operator $\Delta^{-1} : \left\{ u \in L^2 | \int_\Omega u dx = 0 \right\} \to H^2(\Omega)$ denoted

$$f := \Delta^{-1} u$$

with the property that

$$\langle \Delta f, v \rangle_{L^2} = \langle u, v \rangle_{L^2}$$

for any $v \in L^2(\Omega)$. This mapping requires boundary conditions to make it unique (e.g. $\Delta^{-1} : L^2(\Omega) + u|_{\partial\Omega} = 0 \to H^2_{periodic}(\Omega)$). Further, we recall the following equivalent formulations of the H^{-1} inner product when $u, v \in L^2(\Omega)$ satisfy periodic boundary conditions,

$$\langle u, v \rangle_{H^{-1}} = \left\langle \Delta^{-1} u, v \right\rangle_{L^2} = \left\langle u, \Delta^{-1} v \right\rangle_{L^2} \tag{1.4}$$

The global existence of solutions to (1.3) was shown by Elliot and Songmu in 1986 [13]. Solutions to the Cahn-Hilliard equation undergo a rapid spinodal decomposition into domains of the two pure phases ($u = \pm 1$) separated by a boundary layer with thickness of $O(\varepsilon)$. The evolution of the separated phase domains was studied first by Pego, who showed that the motion of the interfaces is a Mullins-Sekerka type flow [11]. Later, asymptotic analysis by Alikakos et al. derived the rigerous convergence of the Cahn-Hilliard equation to the Mullins-Sekerka flow [14]. Further analysis was performed by Modica and Sternberg showing that solutions of the Cahn-Hilliard equation minimize the area of the interface surfaces [15, 16]. Work by de Mottoni and Schatzman, showed that the motion of the developed phase interfaces is driven by the mean curvature of the surface [17].

1.3 Sharp Interfaces and the Canham-Helfrich Model

The limit $\varepsilon \to 0$ corresponds to sharp-interface boundaries between phase domains. A common assumption is that the potential energy stored in such interfaces depends exclusively on the area of the surface and curvature of the interface. For a surface in three dimensions and a point, p, on that surface, consider all the curves on the surface passing through p. Each curve will have a curvature at p defined by $\kappa = \frac{1}{R}$, where R is the radius of the circle defining that curvature. Take the minimal and maximal curvatures over that set of curves to obtain κ_1 and κ_2 respectively. We can then define the mean curvature and Gaussian curvature as

$$H = \frac{1}{2}(\kappa_1 + \kappa_2) \qquad K = \kappa_1 \kappa_2.$$

The Canham-Helfrich free energy is a generic model for the bending of thin elastic films which truncates the dependence upon curvature at quadratic order. Canham derived the model when studying the biconcave shape of human red blood cells [18], and separately, Helfrich developed it to describe the minimum energy geometries of lipid bilayers and vesicles [19]. In three space dimensions, the Canham-Helfrich energy is

$$\mathcal{E}_C(\Gamma) = \int_\Gamma \left[a_1 + a_2 (H - a_3)^2 + a_4 K \right] dS, \tag{1.5}$$

where a_1 is a constant that denotes the energy density per unit of surface area, a_2 and a_4 are the energy densities attributed to the respective curvatures, and a_3 specifies the intrinsic, or zero-energy, value of the mean curvature.

Due to the fact that phase separation is dominated by interfacial energies, the Canham-Helfrich model must be taken into consideration. Du et al. show that the Canham-Helfrich

model is sufficient to generate many important geometrical structures [20], and the Canham-Helfrich energy is considered to be generic [21]. However, the model brings with it some challenges. As a sharp interface model, Canham-Helfrich models do not have the ability to account for topological changes, nor can it predict the dependence of its parameters on the interfacial structure.

To address some of these issues, Gurtin and Jabbour proposed a diffuse interface model that accounts for the smoothing of sharp corners found in grain boundaries of crystalline materials by limiting the energy dependence on curvature to a constitutive framework[22].

1.4 The Functionalized Cahn-Hilliard Model

The Functionalized Cahn-Hilliard (FCH) model was developed by Promislow to describe nanostructure morphology changes in functionalized polymer chains that have been hydrated with a polar solvent [23]. Hydrocarbon backbones of long polymers are hydrophobic, and when they are made into membranes, they exclude any polar solvent such as water. To make such a membrane useful in applications such as Polymer Electrolyte Membrane (PEM) fuel cells, there must be a porous network of thin water channels to allow conduction of protons while excluding electrons [24]. The immiscibility of the water and polymer can be reversed by a functionalization process whereby acid terminated side-chains are added to the polymer backbone. This functionalization embeds latent energy into the membrane that can be released when water is introduced. This energy reduction occurs when the polymer and water phases separate, and a polymer-solvent interface is formed. At the interface, the acid groups can minimize energy by releasing their protons and solvating the bound negative ions.

Most phase field models are obtained from a perturbation about a spatially uniform

density. The FCH model is fundamentally different because it is obtained by perturbing a model which stabilizes preferred geometries. The Functionalized Cahn-Hilliard energy is

$$\mathcal{E}(u) = \int_\Omega \frac{1}{2}\left(\varepsilon^2 \Delta u - W_\tau'(u)\right)^2 - \varepsilon\left(\frac{\varepsilon^2 \eta_1}{2}|\nabla u|^2 + \eta_2 W_\tau(u)\right) dx, \tag{1.6}$$

where the first term in the integral comes from the variational derivative of the Cahn-Hilliard energy (1.1) which stabilizes bilayer, pore, and micelle structures. The second term gives the perturbation which promotes the growth of interface and competition between these geometries.

As with the Cahn-Hilliard energy, the function u takes values from -1 to 1 and defines the volume fraction of the polymer and water phases (1.1). The parameter ε defines the thickness of the boundary layer separating phase domains. η_1 and η_2 are $O(1)$ or $O(\varepsilon)$ constant parameters that govern the nature of the energetic interations. η_1 can be compared to the electrostatic energy of solvating the side chains, and η_2 describes pressure associated with differing mixtures of the two phases.

$W_\tau(u)$ is a double-well potential energy function as before, but it is no longer symmetric due to the difference in self-energies of polymer and water (Figure 1.2). Typically $W_\tau(u) = \frac{1}{2}(u+1)^2\left(\frac{1}{2}(u-1)^2 - \frac{\tau}{3}(u-2)\right)$, where the positive parameter τ defines the amount that the well is tilted. This asymmetry, along with η_1 and η_2, gives rise to various geometries that can each minimize the energy. This behavior is not observed in the CH model with a tilted well because the addition of a linear tilt is eliminated by the H^{-1} gradient flow. In the FCH energy, the slope of the introduced linear term remains hidden in the positive squared term.

Solutions of the FCH equation evolve on time scales ranging from $O\left(\varepsilon^2\right)$ through $O\left(\varepsilon^{-2}\right)$

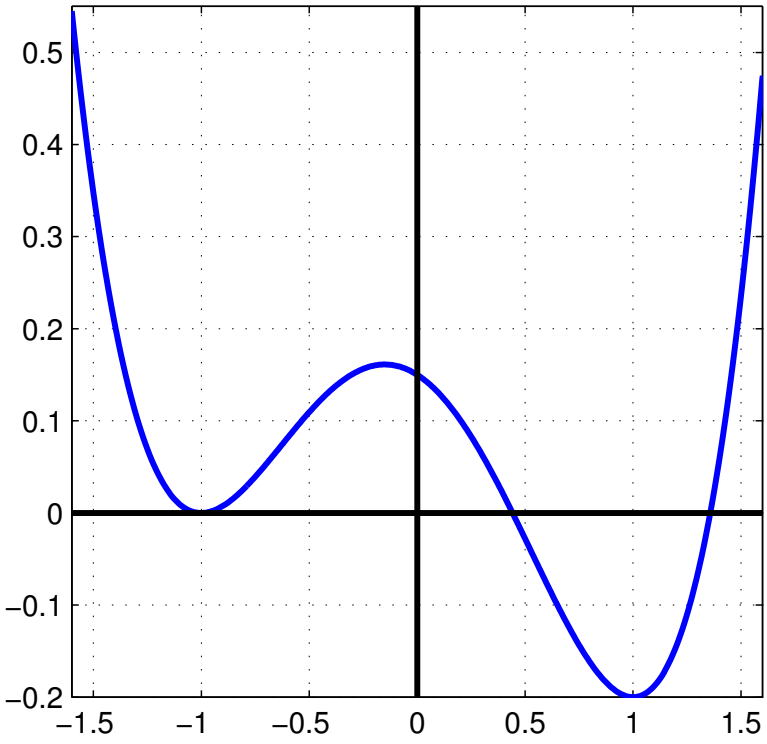


Figure 1.2 Tilted potential energy function $W(u)$

[25]. At the fastest time-scales, $O\left(\varepsilon^2\right)$ and $O\left(\varepsilon^1\right)$, initial data relaxes to form the proper front profiles between domains. When time is $O(1)$, domains form according to a nonlinear diffusion equation which has stable equilibria at the zeros of $W'(u)$, specifically $u(x) = -1$. At longer $O\left(\varepsilon^{-1}\right)$ time-scales, geometric structures evolve with a quenched mean curvature flow for the normal velocity of the domain fronts. Finally, for the longest time evolution, $O\left(\varepsilon^{-2}\right)$, domains that are separated in space compete for the minor phase, $u = +1$. The aim of this work is to accurately capture all of these disparate time-scales.

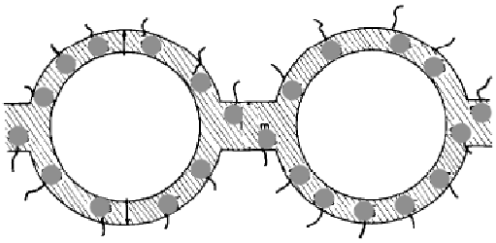
Work by Gompper and Schick used small angle x-ray scattering data (SAXS) to motivate an energy model similar to the FCH model that describes amphiphilic systems such as two immiscible fluids mixed with a surfactant [26, 27]. In their study, they show a strong

connection between the Ginzburg-Landau energy model, the Canham-Helfrich surface energy, and experimental data. This connection can be combined with asymptotic analysis by Gavish et al. that shows convergence of the FCH energy to the Canham-Helfrich energy in the limit as $\varepsilon \to 0$ [28]. Taken together, the analysis and experimental results suggest that the FCH model can be useful in probing the physically relevant geometries created by functionalized polymer/solvent systems.

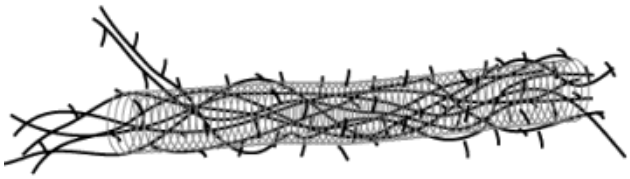
1.5 Proposed Experimental Nanomorphologies

Recently Gavish et al. described the nature of geometric minimizers to the FCH model [28]. In three dimensions, energy minimizing geometries include cylinders, inverted micelles, and bilayers. Experimental and numerical studies of a functionalized polymer, Nafion, have yielded a multitude of predicted geometric structures, some of which are shown in Figure 1.3. Nafion is a membrane made from sulfonated tetrafluoroethylene that was discovered in the 1960s by Walther Grot [33]. It is composed of long, hydrophobic, polymer chains that have been functionalized by attaching short, sulfonate-tipped, side-chains. Hydrolysis of these sulfonate groups can release latent energy, so small domains of water form inside the membrane to lower the free energy. Any continuum model for Nafion must be inherently binary due to the rarity of water embedded in the membrane. Water and low hydration ions form a common phase, so the value of $u(x)$ represents the density of ions in water at each point in the domain.

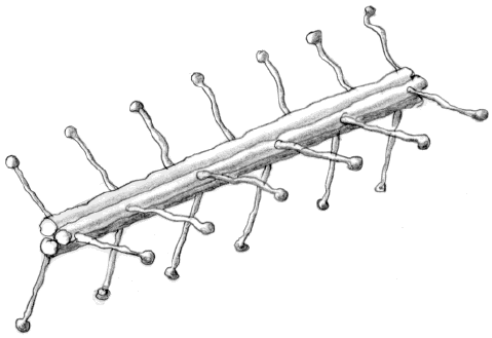
One of the first characterizations of Nafion was performed by Hsu and Gierke who predicted cluster chain pores [29]. A study by Ioselevich et al. attempted to combine experimental data from many experiments, and they believe that the Nafion backbones group to



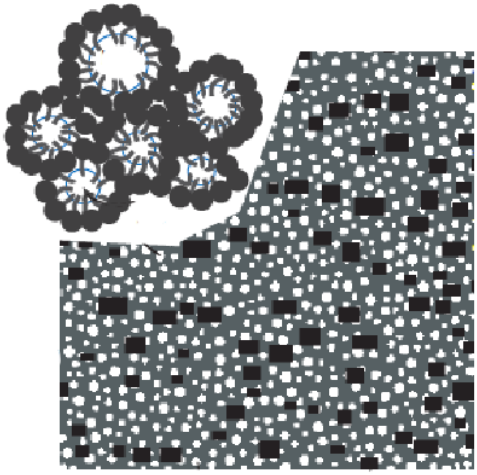
Cluster chain morphology from work by Hsu and Gierke [29]



Cylindrical pore morphology predicted by Rubatat et al. [31]



Nafion backbone groupings suggested by Ioselevich [30]



Morphology fit to SAXS data by Schmidt-Rohr and Chen [32]

Figure 1.3 Proposed Nafion nanomorphologies based on experimental results

provide a higher number of solvated acid groups per unit length [30]. Rubatat et al. used small angle x-ray scattering (SAXS) and small angle neutron scattering (SANS) to probe the structure of Nafion, and they concluded that the water domains were cylindrical in shape [31]. Schmidt-Rohr and Chen predicted parallel cylindrical pores and then numerically adjusted the two dimensional cross-section until it matched SAXS data [32]. From essentially the same experimental data, many different geometries have been predicted. Through simulations on the Functional Cahn-Hilliard equation, we intend to investigate the range of possible nanomorphologies that minimize the FCH energy and provide tools to better describe the balance of curvature and surface energies.

1.6 Functionalized Cahn-Hilliard Equation

Before running simulations on the FCH model, we must convert it into a differential equation. The FCH energy (1.6) can be formally derived from the CH energy (1.1) by assigning a negative value to the interfacial energy via the Cahn-Hilliard energy, $\mathcal{E}_{CH}$, then balancing it against the square of its own variational derivative. Thus the FCH energy can be written as

$$\mathcal{E}(u) = \int_{\Omega} \frac{1}{2} \left(\frac{\delta \mathcal{E}_{CH}}{\delta u} \right)^2 dx - \eta \mathcal{E}_{CH}, \tag{1.7}$$

where we have simplified to the case $\eta = \varepsilon\eta_1 = \varepsilon\eta_2$. Physically this describes electrostatic energy competition. The square of $\frac{\delta \mathcal{E}_{CH}}{\delta u}$ describes phase separation that lowers energy by separating the polar solvent and non-polar polymer. Opposed to this is the negative Cahn-Hilliard term, $-\eta\mathcal{E}_{CH}$, which lowers the energy by solvating the ionic polymer side-chains. Again, we introduce the chemical potential using formula (1.2) and note that the variational derivative of (1.6) can be written in terms of $\frac{\delta \mathcal{E}_{CH}}{\delta u}$ to obtain

$$\begin{aligned} \mu = \frac{\delta \mathcal{E}}{\delta u} &= \left(\frac{\delta^2 \mathcal{E}_{CH}}{\delta u^2} - \eta \right) \frac{\delta \mathcal{E}_{CH}}{\delta u} \\ &= -\left(\varepsilon^2 \Delta - W''(u) - \eta \right) \left(\varepsilon^2 \Delta u - W'(u) \right). \end{aligned} \tag{1.8}$$

If we reintroduce the different values for η_1 and η_2 we obtain

$$\mu = -\left(\varepsilon^2 \Delta - W''(u) - \varepsilon\eta_1 \right) \left(\varepsilon^2 \Delta u - W'(u) \right) - \varepsilon(\eta_1 - \eta_2) W'(u). \tag{1.9}$$

As before, we take the H^{-1} gradient flow on the chemical potential. This gives the working form of the FCH equation,

$$u_t = -\Delta\mu = \Delta\left[\left(\varepsilon^2\Delta - W''(u) - \varepsilon\eta_1\right)\left(\varepsilon^2\Delta u - W'(u)\right) - \varepsilon\left(\eta_1 - \eta_2\right)W'(u)\right]. \quad (1.10)$$

A similar derivation has been shown by Gavish et al. for the Allen-Cahn like version of the equation [28]. That version of the model does not conserve mass, so it must be explicitly accounted for with a zero-mass projection at each step. This is accomplished by subtracting off the average change in solution from every point in the domain.

With the FCH equation we need to specify initial and boundary conditions. The initial condition is simply $u(x,0) = u_0(x)$, where u_0 is a function that identifies the composition of the material at each point in space and varies between -1 (pure polymer) and 1 (pure solvent) over the entire domain, Ω. The boundary conditions can take many forms depending on the set up of the experiment, but for now we use periodic boundary conditions since we are simulating a small domain inside bulk material of the same composition.

1.7 Numerical Challenges for the CH and FCH

The Cahn-Hilliard and Functionalized Cahn-Hilliard equations both have some significant numerical challenges when simulating solutions. When the CH equation (1.3) is written out with the standard double well potential, $W(u) = \frac{1}{4}\left(u^2 - 1\right)^2$, we have the equation:

$$u_t = -\Delta\left(\varepsilon^2\Delta u - u^3 + u\right).$$

This equation is notoriously difficult to approximate numerically due to the small time step size required to maintain stability of the solution. The behavior is often referred to as stiffness in the numerical approximation, and for this equation, it is due to both the harmonic operators and the nonlinear operator [34].

As an example, if the forward Euler method is used and $\varepsilon = 0.01$, the time steps must be on the order of 10^{-7} for the method to be stable. What makes the problem worse is that the fastest evolution in the solution is $O\left(\varepsilon^2\right) \sim 10^{-4}$ and Ostwald ripening (growth of large phase domains at the expense of smaller domains) is on a much slower $O\left(1\right)$ time scale [35]. This means that an explicit method requires far too many time steps to get any reasonable results from a simulation. On the other hand, if backward Euler is used with the hope of obtaining a larger time step, the nonlinearity causes implicit solvers to fail due to large time steps moving the solution out of the basin of attraction (as in Newton's method). The implicit time step restriction is not quite as dramatic as the restriction for an explicit method, but it is still unreasonably inefficient.

When attempting to simulate solutions to the FCH equation (1.10), the set of numerical challenges includes the difficulties from the CH equation, but it also has other challenges as well. The equation includes a third laplacian operator and the tilted potential well increases the complexity of the polynomial terms (Figure 1.2), but the biggest difference is not numerical but physical. In the CH model, the complexity of the solution geometry decreases in time because a decrease in phase interactions leads to a decrease in energy. In the FCH model the solution can also decrease energy by *increasing* the amount interface between phases, so solutions to the FCH equation can become more complex as time evolves. These extra challenges require a researcher to think carefully about the types of methods to use when working on the FCH equation, but methods that have worked for the CH equation are

the best place to start looking for useful ideas.

1.8 Numerical Methods for the CH Equation

Since the FCH model is new, numerical methods to simulate solutions to it have not yet been developed. Development of a time-stepping method for the FCH equation is the purpose of this research. Due to the strong connections between the CH and FCH models, a review of methods for solving the Cahn-Hilliard equation is necessary. A literature search returns far too many papers to discuss here, so this review will only touch on a few papers from each of the most relevant numerical methods. I divide the review into two parts: spatial schemes and time-stepping schemes. Any good numerical method for simulating the CH or FCH equation will require both a spatial and temporal component, and the connections between the two could be very important to the performance of the method.

1.8.1 Spatial Schemes

The most common numerical method for solving the Cahn-Hilliard equation (1.3) has been the method of finite elements. Elliot and French were active in this area of research in the late 1980s. In their research, they developed several different finite element methods, each with slightly different properties. In the first paper, they present a method for solving the CH equation in one dimension and discuss the severity of the time step restriction due to stability [36]. Two years later with Milner, they propose a method that is second order in space and only requires the elements to be continuous [37]. In a second paper in 1989, Elliot and French present a non-conforming finite element approach and give proofs for the accuracy and convergence of the method [38].

In 1991, Du and Nicolaides presented a semi-discrete finite element method that has the advantage of a Lyapunov function [39]. The existence of a Lyapunov function for their method makes it possible to prove convergence of the approximate solutions without any conditions beyond the existence and uniqueness of solutions to the original differential equation. Later, Barret et al. used a finite element method to model solutions to the CH equation with non-constant surface diffusion [40]. In the last decade, Feng and Prohl developed and analyzed a mixed finite element scheme, and they proved some error bounds that were $O\left(\varepsilon^{-1}\right)$, rather than $O\left(\exp\left(\frac{1}{\varepsilon}\right)\right)$ which was the previous result [41].

A common thread through several of the methods, including the finite element schemes, was the use of multigrid techniques. In 2006, Kay and Welford gave a multigrid finite element scheme with mesh independent convergence rates [42]. It is interesting to note that this paper also discusses the difficulty of time step restrictions. An earlier paper by Kim et al. uses a conservative multigrid method to solve the CH equation. The solution is then coupled to a projection method to solve fluid flow with the Navier-Stokes equation for a two fluid system [43].

In the last few years, several papers have been published by Wise and his colleagues using non-linear multigrid finite difference schemes on the Phase Field Crystal model [7, 44, 45]. The Phase Field Crystal model is an extension of the Cahn-Hilliard model that has strong anisotropy so that it can effectively model crystal growth and formation. Their schemes are unconditionally stable with respect to time step size because they use a convex splitting technique developed by Eyre [46, 34]. The technique will be used for the method developed in Chapter 2, and it will be discussed extensively there. Previous to the Wise papers, Furihata published a finite difference scheme that was second order in time and space but suffered from the severe time step conditions on stability [47].

Spectral methods have also played an important role in simulating solutions to the CH equation. In 1999, Zhu et al. applied a semi-implicit spectral method to the CH equation to study the long time behavior in a two dimensional domain [48]. In their scheme, they treated the principle elliptic operator implicitly and all the other terms explicitly. Later, Vollmayr-Lee and Rutenburg used a spectral method to identify an $O\left(t^{\frac{1}{3}}\right)$ time step size which can control accuracy for a certain class of convex splitting schemes [49]. They came to the conclusion based on Eyre's Theorem [46] and von Neumann analysis. In 2005, Ye and Cheng published a paper discussing the inheritance of energy dissipation and mass conservation in spectral methods [50]. Recently, Shen and Yang published the results of error estimates for several methods used to simulate the CH equation, and in their conclusion, they recommend spectral methods because of their effectiveness in capturing the interface fronts in the solution [51].

Lastly, discontinuous Galerkin methods have been applied to the CH equation with some success. In 2006, Wells et al. discussed a discontinuous Galerkin scheme and compared the results and convergence to a standard finite element method [52]. A year later, Xia et al. presented a local discontinuous Galerkin scheme that comes with a proof of energy stability [53]. The paper includes numerical accuracy results and also some results on a ternary system which could be of interest to our group in future research.

1.8.2 Temporal Schemes

Chapters 2, 3, and 4 will discuss in depth three temporal schemes for the Functionalized Cahn-Hilliard equation: convex splitting, fully implicit, and exponential time differencing. Before delving into the work of this dissertation, I will review these three temporal methods for the Cahn-Hilliard equation and other partial differential equations.

In 1998, Eyre proposed a gradient splitting method for stabilizing the numerical computation of equations that have the form $u_t = F(u) = \Delta \frac{\delta \mathcal{E}}{\delta u}$. Eyre applied this method to the Cahn-Hilliard equation (1.3) to take advantage of the convexity of the associated Cahn-Hilliard energy (1.1). By appealing to convexity, the implicit portion of the discretized equation discussed in Section 1.7 will have a unique solution. This small adjustment in the method completely removes the time step restriction. The method is discussed in two papers, but only one of them was actually published [34, 46].

Several other authors have used convex splitting of the CH energy or similar models to improve performance of their numerical simulations. These methods are valuable because the convex splittings derived guarantee a unique solution at each time step and provide for methods that decay in energy at each step. In 2009, Wise, Wang, and Lowengrub presented an energy stable scheme for the phase field crystal equation [54]. Following this in 2010, Wise used the same scheme to approximate solutions to the Cahn-Hilliard-Hele-Shaw system of equations in a nonlinear multigrid framework [45]. Later, Gomez and Hughes developed a convex splitting based on the fourth derivative of $W(u)$ that attains second order convergence in time [55].

A method that does not fall into the convex splitting category but retains gradient stability is the implicit-explicit stabilized scheme. Stabilization comes from adding and subtracting a term that stabilizes inversion of the linear portion of the operator. Papers in 2010 by Shen and Yang develop a first order version of the method [56] and review several other energy stable methods [51]. A later paper in 2012 by Shen, Wang, and Wang extends their analysis to a gradient stable scheme that is second order accurate in time.

Before the development of gradient stable schemes, simulation of the CH equation was primarily performed with implicit schemes since explicit computations suffer from severe

time step restrictions. An early paper on computing the CH equation with implicit time stepping in one dimension comes from Elliot and French [36]. Further work with Milner provided a second order in space calculation [37], and French alone published a paper on an implicit scheme for the CH equation in two dimensions [57]. Lastly, a paper by Du and Nicolaides established a scheme with better stability properties [39].

More recently in 2008, Kronbichler and Kreiss published a paper on two phase flows where they used implicit time stepping for the Cahn-Hilliard portion [58]. In 2011, Willoughby completed his doctoral dissertation with Brian Wetton on implicit time stepping methods for the Allen-Cahn and Cahn-Hilliard equations [59].

The third method I implement in this dissertation is the exponential time differencing (ETD) method. ETD methods for ordinary differential equations originated as early as 1960 in work by Certaine [60] and Pope [61]. In the nineteen-sixties and seventies, A great many papers followed that refined the idea, but it eventually fell out of favor because it was infeasible on the current computing hardware. An extensive review of ETD and other similar methods was published by Minchev and Wright [62].

With recent advances in computing hardware and parallelism, ETD methods have finally become effective for partial differential equations. In 1994, Hou, Lowengrub, and Shelley used an exponential linearization to remove the stiffness from the computation of interfacial flows with surface tension [63]. This lead to a rash of papers being published within the last fifteen years on the topic of ETD methods for PDEs. In 1998, Beylkin, Keiser, and Vozovoi develloped discretization schemes for nonlinear PDEs [64]. Four years later, Cox and Matthews expanded the schemes to include higher order Runge-Kutta methods [65]. In 2004 and 2005, Du and Zhu published a pair of papers analyzing the stability of ETD-Runge-Kutta (ETD-RK) methods and proposing a complex integration for stabilizing higher order

terms [66, 67]. Kassam completed his doctoral dissertation on high order time-stepping for semilinear PDEs and published a paper with Trefethen [68, 69]. In 2005, Hochbruck and Ostermann published a pair of papers on ETD-RK methods for parabolic problems [70, 71]. They later published a review paper on "the construction, analysis, implementation and application of exponential integrators" [72], where they focus on the two dominant types of stiffness; equations where the Jacobian has eigenvalues with large negative real parts, or highly oscillatory problems with large, purely-imaginary eigenvalues. Tokman has published a series of papers focused on efficient computation of the exponential coefficients that arise in ETD schemes, particularly how to compute the coefficients for large scale computing [73, 74, 75]. Most recently, Koskela and Ostermann published a paper on extending exponential time differencing to higher orders using Taylor expansion [76].

1.9 Spectral Method

To effectively study temporal schemes we employ a Fourier spectral method in space. We consider a standard Fourier pseudo-spectral discretization of (1.3). The function u is approximated by the vector $\mathbf{U}$ defined on a discrete grid of N points ($x_j = jh$, $j = 1, \dots N$) equally spaced with spacing $h = 2\pi/N$. The discrete Fourier transform of $\mathbf{U}$ is denoted by $\hat{\mathbf{U}}$:

$$\hat{\mathbf{U}}_n = \frac{1}{N} \sum_{j=1}^{N} e^{-i(j-1)(n-1)/N} U_j$$

which we can write in vector form as

$$\hat{\mathbf{U}} = F\mathbf{U}$$

where F is the matrix representation of the discrete Fourier transform. In the pseudo-spectral setting, $\mathbf{U}$ can also be considered to approximate u in the sense

$$u(x,t) \approx \sum_{n=1}^{N} \hat{\mathbf{U}}_n(t) e^{i2\pi\alpha_n x}$$

where α_n are the appropriately aliased n values. We then use the usual spectral approximations for derivatives on grid points

$$u_{xx} \approx F^{-1}\Lambda_\alpha F\mathbf{U} := \Delta_h \mathbf{U}$$

where Λ_α is the diagonal matrix with entries $-\alpha_n^2$. The inverse discrete Fourier matrix $F^{-1} = NF^*$. Note that the matrix Δ_h can be formed explicitly [77] but we only need the property that it is possible to multiply by it efficiently using the FFT.

With these approximations, we can write a method of lines (MOL) discretization (a semi-discretization in space but keeping time continuous) of (1.3):

$$\frac{d\mathbf{U}}{dt} = -\epsilon^2 \Delta_h \Delta_h \mathbf{U} + \Delta_h W'(\mathbf{U}). \tag{1.11}$$

where by $W'(\mathbf{U})$ we mean the vector with entries $W'(\mathbf{U}_j)$.

Throughout this document we will focus on the approximation to the time-stepping and treat the analysis in a semi-discretized form. All of the computer codes are discretized in space using this Fourier spectral decomposition, and the operators are built explicitly in the frequency domain to obtain efficiency.

Chapter 2

Convex Splitting for the FCH Equation

2.1 Gradient Splitting Method

We begin by giving a detailed description of Eyre's convex splitting method. If an energy, $\mathcal{E}(u)$, is strictly convex, then its H^{-1} gradient is contractive. By definition, a functional, $F(u)$, is weakly contractive in the H^{-1} Sobolev space if it satisfies

$$\langle F(u) - F(v), u - v \rangle_{H^{-1}} \leq 0,$$

for all $u, v \in H^2(\Omega)$. Using integration by parts on the H^{-1} inner product with periodic boundary conditions, this is equivalent to

$$\langle \nabla (F(u) - F(v)), \nabla (u - v) \rangle_{L^2} \leq 0, \tag{2.1}$$

Such a function gives energy decay in the approximation to differential equation, i.e.

$$\frac{U^{n+1} - U^n}{k} = F\left(U^{n+1}\right) \Rightarrow \mathcal{E}\left(U^{n+1}\right) < \mathcal{E}(U^n), \tag{2.2}$$

where U^{n+1} and U^n are the numerical solutions and times t_{n+1} and t_n respectively, and k is the size of the time step. On the other hand, if the negated energy, $-\mathcal{E}(u)$, is strictly convex, then $F(u)$ is expansive, we have $\langle F(u) - F(v), u - v\rangle_{H^{-1}} \geq 0$, which gives

$$\frac{U^{n+1} - U^n}{k} = -F(U^n) \Rightarrow \mathcal{E}\left(U^{n+1}\right) < \mathcal{E}(U^n). \tag{2.3}$$

Neither the Cahn-Hilliard energy (1.1) nor the Funcionalized Cahn-Hilliard energy (1.6) are convex, however, they can be split into convex and concave terms. After the separation of energy terms, we obtain the splitting of the gradient function into

$$F(u) = F_c(u) - F_e(u). \tag{2.4}$$

$F_c(u)$ and $F_e(u)$ are the respective contractive and expansive parts of the function $F(u)$. With the gradient function split, Eyre proved that the scheme

$$U^{n+1} - U^n = k\left(F_c\left(U^{n+1}\right) - F_e(U^n)\right) \tag{2.5}$$

is consistent, gradient stable for any $k > 0$, and possesses a unique solution for each time step.

The ability to use any size of time step is a massive improvement over the previous restrictions, but the method is only first order in time. We believe that we could improve the method by applying a deferred correction step to the time updates [78]. The essence of the method is that we take an explicit step on the expansive terms, and it comes with a large amount of error, but the implicit step on the contractive terms makes up for the instability of the explicit portion and keeps the solution decaying in the energy landscape.

2.2 Gradient Splitting Example

To better understand the gradient splitting method, take the Cahn-Hilliard equation in one dimension,

$$\frac{du}{dt} = -\varepsilon^2 u_{xxxx} + \left(u^3\right)_{xx} - u_{xx}.$$

The first terms is contractive since it come from a convex term in the energy, $\frac{\varepsilon^2}{2}|\nabla u|^2$. and respectively. Using (2.1), we can show that it is contractive as follows,

$$\begin{aligned} -\varepsilon^2 \left\langle \partial_x \left(u_{xxxx} - v_{xxxx}\right), \partial_x (u-v) \right\rangle_{L^2} &= -\varepsilon^2 \left\langle \partial_x^5 (u-v), \partial_x (u-v) \right\rangle_{L^2} \\ &= -\varepsilon^2 \left\langle \partial_x^3 (u-v), \partial_x^3 (u-v) \right\rangle_{L^2} = -\varepsilon^2 \left\| \partial_x^3 (u-v) \right\|_2^2 \leq 0 \end{aligned} \quad (2.6)$$

The second term also comes from a convex term in the energy, namely $\frac{1}{4}u^4$, and can be shown to be contractive in a similar fashion.

$$\begin{aligned} \left\langle \partial_x \left(\left(u^3\right)_{xx} - \left(v^3\right)_{xx} \right), \partial_x (u-v) \right\rangle_{L^2} &= \left\langle \partial_x \left(\left(u^3 - v^3\right)_{xx} \right), \partial_x (u-v) \right\rangle_{L^2} \\ &= \left\langle \partial_x^3 \left(u^3 - v^3\right), \partial_x (u-v) \right\rangle_{L^2} = -\left\langle u^3 - v^3, \partial_x^4 (u-v) \right\rangle_{L^2} \\ &= -\left\langle (u-v)\left(u^2 + uv + v^2\right), \partial_x^4 (u-v) \right\rangle_{L^2} \leq -\left\langle u - v, \partial_x^4 (u-v) \right\rangle_{L^2} \\ &= -\left\langle \partial_x^2 (u-v), \partial_x^2 (u-v) \right\rangle_{L^2} = -\left\| \partial_x^2 (u-v) \right\|_2^2 \leq 0 \end{aligned}$$

The third term is the backward heat operator and is expansive since it comes from $-\frac{1}{2}u^2$. We can show that it is expansive by again using (2.1),

$$\begin{aligned} -\left\langle \partial_x \left(u_{xx} - v_{xx}\right), \partial_x \left(u - v\right)\right\rangle_{L^2} &= -\left\langle \partial_x^3 \left(u - v\right), \partial_x \left(u - v\right)\right\rangle_{L^2} \\ &= \left\langle \partial_x^2 \left(u - v\right), \partial_x^2 \left(u - v\right)\right\rangle_{L^2} = \left\| \partial_x^2 \left(u - v\right)\right\|_2^2 \geq 0 \end{aligned} \quad (2.7)$$

Thus we can split the right hand side into (2.4), where the contractive and expansive functions are

$$\begin{aligned} F_c\left(u\right) &= -\varepsilon^2 u_{xxxx} + \left(u^3\right)_{xx} \\ F_e\left(u\right) &= u_{xx}. \end{aligned}$$

Using (2.5), the gradient splitting scheme is

$$U^{n+1} - U^n = k\partial_x^2 \left(-\varepsilon^2 \partial_x^2 U^{n+1} + \left(U^{n+1}\right)^3 - U^n\right) \quad (2.8)$$

To see the nonlinear Newton solve needed, rewrite the equation as

$$U^{n+1} - k\partial_x^2 \left(-\varepsilon^2 \partial_x^2 U^{n+1} + \left(U^{n+1}\right)^3\right) = \left(1 - k\partial_x^2\right) U^n.$$

For each time step, this equation can be solved by calculating the right hand side from the solution at the previous time step, then using Newton's method to iteratively solve the nonlinear implicit portion, and thereby obtaining the solution at the next time step. Due to the contractive nature of the implicit terms, Newton's method is now guaranteed to converge for any time step size, $k > 0$, as discussed in Sections 1.7 and 2.1.

2.3 Splitting the FCH Equation

The gradient splitting method depends on the choice of mixing potential function $W(u)$, since a different choice would give a different collection of convex and concave terms. The tilted, double-well potential used in this research is

$$W(u) = \frac{1}{2}(u+1)^2\left(\frac{1}{2}(u-1)^2 + \frac{\tau}{3}(u-2)\right) \tag{2.9}$$

and is shown in Figure 1.2. The parameter τ governs the amount of tilting in the double-well and has a significant impact on the possible minimal energy geometries.

With the potential substituted into the FCH equation (1.10), we obtain the full PDE:

$$\begin{aligned} u_t = &-\varepsilon^4\Delta^3 u + \varepsilon^2\Delta^2\left(u^3\right) + \frac{1}{2}\varepsilon^2\tau\Delta^2\left(u^2\right) - \varepsilon^2(2+\varepsilon\eta_1)\Delta^2 u + 3\varepsilon^2\Delta\left(u^2\Delta u\right) + \varepsilon^2\tau\Delta(u\Delta u) \\ &- \Delta\left[3u^5 + \frac{5}{2}\tau u^4 + \left(\frac{1}{2}\tau^2 - 4 - \varepsilon\eta_2\right)u^3 - \frac{\tau}{2}(6+\varepsilon\eta_2)u^2 + \left(1+\varepsilon\eta_2 - \frac{1}{2}\tau^2\right)u\right] \end{aligned} \tag{2.10}$$

Considering this equation with a first order approximation of the time derivative, most of the terms on the right hand side of the equation can be classified as contractive or expansive gradients of their corresponding contribution in the energy functional (Equation 1.6). One serious exception is the term $\epsilon^2\Delta\left(\Delta u^3 + 3u^2\Delta u\right)$, which is from the Euler-Lagragian variation of the energy component $\int_\Omega -\epsilon^2 u^3\Delta u d\Omega$. This energy term is neither convex nor concave. Note that $3u^2\Delta u + \Delta u^3 = -6u|\nabla u|^2 + 6\nabla\cdot\left(u^2\nabla u\right)$. To guarantee gradient stability for the scheme, we must adopt the semi-implicit terms $-6u^{n+1}|\nabla u^n|^2 + 6\nabla\cdot\left(\left(u^{n+1}\right)^2\nabla u^{n+1}\right)$ while classifying the other terms as previously discussed. This gives the unconditionally

gradient stable nonlinear semi-implicit scheme

$$\frac{u^{n+1}-u^n}{k} = \Delta\left[\left(4+\epsilon^2\eta\right)(u^n)^3 - \left(2\epsilon^2+\epsilon^4\eta\right)\Delta u^n - \epsilon^4\Delta^2 u^{n+1} - 6\epsilon^2 u^{n+1}\left|\nabla u^n\right|^2 \right.$$
$$\left. +6\epsilon^2\nabla\cdot\left(\left(u^{n+1}\right)^2\nabla u^{n+1}\right) - \left(3\left(u^{n+1}\right)^5 + \left(1+\epsilon^2\eta\right)u^{n+1}\right)\right]. \tag{2.11}$$

Unfortunately, the mixed implicit-explicit terms make the numerical scheme more complicated and significantly decrease its speed and efficiency. We therefore chose to treat the difficult term fully implicitly and include it in $F_c(u)$. The additive splitting can then be completed term by term and substituted into Equation 2.5. This gives the nearly contractive function

$$F_c(u) = -\Delta\left[\varepsilon^2\Delta\left(\varepsilon^2\Delta u - u^3\right) - \varepsilon^2\left(3u^2+\tau u\right)\Delta u + 3u^5 \right.$$
$$\left. -\frac{\tau}{2}\left(6+\varepsilon\eta_2\right)u^2 + \left(1+\varepsilon\eta_2 - \frac{1}{2}\tau^2\right)u\right], \tag{2.12}$$

and the corresponding expansive function

$$F_e(u) = \Delta\left[\varepsilon^2\Delta\left(\left(2+\varepsilon\eta_1\right)u - \frac{1}{2}\tau u^2\right) + \frac{5}{2}\tau u^4 + \left(\frac{1}{2}\tau^2 - 4 - \varepsilon\eta_2\right)u^3\right]. \tag{2.13}$$

Even though we cannot prove that this splitting is guaranteed to give energy decay for every time step size, every simulation we have performed with this method has demonstrated a decrease in energy. This slight deviation from Eyre's original method seems to have little impact on the numerical stability of the scheme.

2.4 Stability and Solvability

First, we will prove that the semi-discretized scheme is indeed unconditionally gradient stable. The proof is algebraic, relying simply on the integration by parts and Young's inequality. We will prove stability for the simpler version of 2.11 which is the L^2 flow on the energy,

$$\frac{u^{n+1}-u^n}{k} = \left(4+\epsilon^2\eta\right)(u^n)^3 - \left(2\epsilon^2+\epsilon^4\eta\right)\Delta u^n - \epsilon^4\Delta^2 u^{n+1} - 6\epsilon^2 u^{n+1}\left|\nabla u^n\right|^2 + 6\epsilon^2\nabla\cdot\left(\left(u^{n+1}\right)^2\nabla u^{n+1}\right) - \left(3\left(u^{n+1}\right)^5 + \left(1+\epsilon^2\eta\right)u^{n+1}\right). \quad (2.14)$$

Theorem 2.4.1. *The semi-implicit scheme (2.14) is unconditionally gradient stable for any time step size k, i.e. $F(u^{n+1}) \le F(u^n)$*

Proof. Choose the test function $\phi = -\left(u^{n+1}-u^n\right)$, multiply it to both sides of (2.14), and integrate over the domain. On the left side, we have $-\frac{1}{k}\left\|u^{n+1}-u^n\right\|^2 \le 0$ which approximates the theoretical decay rate $-k\left\|u_t\right\|^2$. Here and after, we use $||\cdot||$ for standard L^2 norm in the inner product space. On the right side, we will recover the form $F\left(u^{n+1}\right) - F\left(u^n\right) + R$ with $R \ge 0$. This is computed term by term beginning with the linear terms.

- $-\epsilon^4\Delta^2 u^{n+1}$: we have $\left\langle \epsilon^4\Delta u^{n+1}, \Delta\left(u^{n+1}-u^n\right)\right\rangle$, which equals

$$\frac{\epsilon^4}{2}\left(\left\|\Delta u^{n+1}\right\|^2 - \left\|\Delta u^n\right\|^2 + \left\|\Delta u^{n+1}-\Delta u^n\right\|^2\right). \quad (2.15)$$

- $-\left(1+\epsilon^2\eta\right)u^{n+1}$: we have $\left\langle\left(1+\epsilon^2\eta\right)u^{n+1}, u^{n+1}-u^n\right\rangle$, which is

$$\frac{\left(1+\epsilon^2\eta\right)}{2}\left(\left\|u^{n+1}\right\|^2 - \left\|u^n\right\|^2 + \left\|u^{n+1}-u^n\right\|^2\right). \quad (2.16)$$

- $-\left(2\epsilon^2+\epsilon^4\eta\right)\Delta u^n$: after integrating by parts, it reads $\left\langle\left(2\epsilon^2+\epsilon^4\eta\right)\nabla u^n,\nabla\left(u^n-u^{n+1}\right)\right\rangle$, and therefore can be written as

$$\left(\epsilon^2+\frac{1}{2}\epsilon^4\eta\right)\left(\|\nabla u^n\|^2-\left\|\nabla u^{n+1}\right\|^2+\left\|\nabla\left(u^n-u^{n+1}\right)\right\|^2\right). \tag{2.17}$$

- $-3\left(u^{n+1}\right)^5$: it reads $\left\langle 3\left(u^{n+1}\right)^5,u^{n+1}-u^n\right\rangle$. We shall show that

$$\left\langle 3\left(u^{n+1}\right)^5,u^{n+1}-u^n\right\rangle\geq\frac{1}{2}\left\langle 1,\left(u^{n+1}\right)^6-(u^n)^6\right\rangle. \tag{2.18}$$

  Define a new function

$$f\left(u^{n+1}\right)=3\left(u^{n+1}\right)^5\left(u^{n+1}-u^n\right)-\frac{1}{2}\left(\left(u^{n+1}\right)^6-(u^n)^6\right).$$

  We show that $f\left(u^{n+1}\right)\geq 0$ by first noting that $f\left(u^n\right)=0$. Further, $f'\left(u^{n+1}\right)=15\left(u^{n+1}\right)^4\left(u^{n+1}-u^n\right)$ which is non-negative for $u^{n+1}>u^n$, non-positive for $u^{n+1}<u^n$. Thus (2.18) holds.

- $\left(4+\epsilon^2\eta\right)(u^n)^3$: a simple calculation gives a result similar to (2.18), i.e. $(u^n)^3\left(u^n-u^{n+1}\right)\geq\frac{1}{4}\left((u^n)^4-(u^{n+1})^4\right)$. Thus

$$\left\langle\left(4+\epsilon^2\eta\right)(u^n)^3,u^n-u^{n+1}\right\rangle\geq\left(1+\frac{1}{4}\epsilon^2\eta\right)\left\langle 1,(u^n)^4-\left(u^{n+1}\right)^4\right\rangle. \tag{2.19}$$

- $-6u^{n+1}\left|\nabla u^n\right|^2+6\nabla\cdot\left(\left(u^{n+1}\right)^2\nabla u^{n+1}\right)$: note that for the first term we have

$$6u^{n+1}\left|\nabla u^n\right|^2\left(u^{n+1}-u^n\right)\geq 3\left(u^{n+1}\right)^2\left|\nabla u^n\right|^2-3\left(u^n\right)^2\left|\nabla u^n\right|^2,$$

therefore

$$\left\langle 1, 6u^{n+1}\left|\nabla u^{n}\right|^{2}\left(u^{n+1}-u^{n}\right)\right\rangle \geq \left\langle 1, 3\left(u^{n+1}\right)^{2}\left|\nabla u^{n}\right|^{2}-3\left(u^{n}\right)^{2}\left|\nabla u^{n}\right|^{2}\right\rangle. \quad (2.20)$$

The second term gives

$$\left\langle 6\left(u^{n+1}\right)^{2}, \nabla u^{n+1}\left(\nabla u^{n+1}-\nabla u^{n}\right)\right\rangle \geq \left\langle 1, 3\left(u^{n+1}\right)^{2}\left|\nabla u^{n+1}\right|^{2}-3\left(u^{n+1}\right)^{2}\left|\nabla u^{n}\right|^{2}\right\rangle. \quad (2.21)$$

Adding (2.21) to (2.20), we obtain

$$\begin{aligned}\epsilon^{2}\left\langle -6u^{n+1}\left|\nabla u^{n}\right|^{2}+6\nabla\cdot\left(\left(u^{n+1}\right)^{2}\nabla u^{n+1}\right), -\left(u^{n+1}-u^{n}\right)\right\rangle \geq \\ \epsilon^{2}\left\langle 1, 3\left(u^{n+1}\right)^{2}\left|\nabla u^{n+1}\right|^{2}\right\rangle-\epsilon^{2}\left\langle 1, 3\left(u^{n}\right)^{2}\left|\nabla u^{n}\right|^{2}\right\rangle. \quad (2.22)\end{aligned}$$

From (2.15 - 2.22), we conclude that $F\left(u^{n+1}\right) \leq F\left(u^{n}\right)$, with $F(u)$ defined by (1.6). □

Next, we will show that there is at most one solution to the nonlinear equation (2.14) for any time step k, therefore, there shall be no restriction from the solvability of (2.14)

Theorem 2.4.2. *The nonlinear equation (2.14) has only one solution in $H^2(\Omega)$ for any time step k.*

Proof. Assuming $u^{n+1} = w, v$ are distinct solutions, by subtraction it follows that

$$\begin{aligned}\frac{w-v}{k} = -\epsilon^{4}\Delta^{2}(w-v)-\epsilon^{2}\left(6w\left|\nabla u^{n}\right|^{2}-6v\left|\nabla u^{n}\right|^{2}\right)+6\epsilon^{2}\nabla\cdot w^{2}\nabla w \\ -6\epsilon^{2}\nabla\cdot v^{2}\nabla v-\left(3w^{5}+\left(1+\epsilon^{2}\eta\right)w-3v^{5}-\left(1+\epsilon^{2}\eta\right)v\right). \quad (2.23)\end{aligned}$$

Multiplying both sides of (2.23) by $w-v$ and integrating over the domain, the left hand side of (2.23) becomes $\left\langle \frac{1}{k}, (w-v)^2 \right\rangle \geq 0$. It suffices to show that the right hand side of (2.23) is non-positive, thus $w = v$ in $H^2(\Omega)$. The right hand side gives

$$\begin{aligned} &-\epsilon^4 \|\Delta(w-v)\|^2 - 6\epsilon^2 \left\langle |\nabla u^n|^2, (w-v)^2 \right\rangle - \left(1+\epsilon^2\eta\right)\|w-v\|^2 \\ &- \left\langle 3\left(w^5 - v^5\right), (w-v) \right\rangle + \left\langle 6\epsilon^2 \nabla\cdot(w)^2 \nabla(w) - 6\epsilon^2 \nabla\cdot(v)^2 \nabla(v), w-v \right\rangle. \end{aligned}$$

It suffices to show that

$$\begin{aligned} g(w,v) = &-\epsilon^4 \|\Delta(w-v)\|^2 - \left\langle 3\left(w^5 - v^5\right), w-v \right\rangle \\ &+ \left\langle 6\epsilon^2 \nabla\cdot w^2 \nabla w - 6\epsilon^2 \nabla\cdot v^2 \nabla v, w-v \right\rangle \leq 0. \end{aligned} \tag{2.24}$$

It is easy to check that $3\left(w^5 - v^5\right)(w-v) \geq \left(w^3 - v^3\right)^2$, therefore

$$g(w,v) \leq -\epsilon^4 \|\Delta(w-v)\|^2 - \left\|w^3 - v^3\right\|^2 + \left\langle 6\epsilon^2 \nabla\cdot w^2 \nabla w - 6\epsilon^2 \nabla\cdot v^2 \nabla v, w-v \right\rangle. \tag{2.25}$$

Integration by parts gives

$$\begin{aligned} \left\langle 6\epsilon^2 \nabla\cdot w^2 \nabla w - 6\epsilon^2 \nabla\cdot v^2 \nabla v, w-v \right\rangle &= -\epsilon^2 \left\langle 6w^2 \nabla w - 6v^2 \nabla v, \nabla(w-v) \right\rangle \\ &= 2\left\langle w^3 - v^3, \epsilon^2 \Delta(w-v) \right\rangle. \end{aligned}$$

Using Young's inequality, it follows that

$$\left\langle 6\epsilon^2 \nabla\cdot w^2 \nabla w - 6\epsilon^2 \nabla\cdot v^2 \nabla v, w-v \right\rangle \leq \epsilon^4 \|\Delta(w-v)\|^2 + \left\|w^3 - v^3\right\|^2. \tag{2.26}$$

Adding (2.26) to (2.25), we have $g(w,v) \leq 0$, therefore $\|w-v\|^2 = 0$, i.e., $w = v$. $\square$

So far, after assuming existence and regularity of the solution to (2.14), we have shown that the semi-discrete scheme (2.14) is unconditionally gradient stable and admits at most one solution. We would like to comment that

Remark 1. *When the finite element method with solution base $H_h^2(\Omega)$ is used, the above analysis holds with suitable boundary conditions. However, Theorem 2.4.1 and 2.4.2 may not be true for other spatial discretization. The motivation of designing such a scheme is due to the fact that one of the major difficulties for the numerical simulation of (1.10) is an accurate time stepping strategy.*

Remark 2. *The generalization of this proof to the conservative H^{-1} flow (2.11) in the semi-discrete framework is straightforward by using the test function $\phi = \Delta^{-1}\left(u^{n+1} - u^n\right)$.*

2.5 Fixed-Point Iteration

To solve the implicit portion of equation (2.5), we apply Newton's method

$$G'\left(U^r\right)\left(U^{r+1} - U^r\right) = -G\left(U^r\right), \tag{2.27}$$

where r is the iteration number, and we define

$$G\left(U^{n+1}\right) = U^{n+1} - U^n - kF_c\left(U^{n+1}\right) + kF_e\left(U^n\right). \tag{2.28}$$

We desire to iteratively solve for U^r such that $G\left(U^r\right) < \epsilon$ with ϵ a desired tolerance. First we calculate the Fréchet derivative

$$G'\left(U^r\right) v = v - kF_c'\left(U^r\right) v, \tag{2.29}$$

with peturbation v and F_c' defined as

$$\begin{aligned} F_c'\left(U^r\right) v = -\Delta \Big[\varepsilon^2 \Delta \left(\varepsilon^2 \Delta v - 3\left(U^r\right)^2 v \right) - \varepsilon^2 \left(6U^r v + \tau v\right) \Delta U^r - \varepsilon^2 \left(3\left(U^r\right)^2 + \tau U^r \right) \Delta v \\ +15\left(U^r\right)^4 v - \tau\left(6 + \varepsilon\eta_2\right) U^r v + \left(1 + \varepsilon\eta_2 - \frac{1}{2}\tau^2 \right) v \Big]. \end{aligned} \tag{2.30}$$

The Fréchet derivative is necessary in this use of Newton's method because $G\left(u\right)$ is a functional on $H^6\left(\Omega\right)$. The Fréchet derivative of a function $f\left(u\right)$ is defined as the linear operator $A\left(u\right)$ that satisfies the following limit,

$$\lim_{\|v\| \to 0} \frac{\|f\left(u+v\right) - f\left(u\right) - A\left(u\right) v\|}{\|v\|} = 0.$$

It can be easily calculated by computing $f\left(u+v\right) - f\left(u\right)$ and then keeping only the terms that are linear in v. This is precisely how we obtained (2.30) from (2.12).

It is important to note in equation (2.29) that $G'\left(U^r\right)$ is a complicated function of U^r. Computing this operator for every step of the iterative solve is computationally infeasible. Instead, we approximate G' with the operator

$$\tilde{G}' = 1 + k\Delta \left[\varepsilon^4 \Delta^2 - c_2 \varepsilon^2 \Delta + c_1 + \left(1 + \varepsilon\eta_2 - \frac{1}{2}\tau^2 \right) \right] \tag{2.31}$$

which is independent of the phase-field solution U^r, and be computed entirely spectrally

which is a very quick calculation. c_1 and c_2 are $O(1)$ empirical constants that approximate the maximum values of the terms depending on U^r. With this approximation, the new fixed-point method loses the second order convergence of Newton's method, but the total computational cost is much lower for each iterative process.

We do not yet have a proof that this splitting applied to the FCH equation (1.10) is a contraction mapping, so we will present a proof that is applied to the Cahn-Hilliard equation (1.3) [79]. The proof is in a finite element framework, even though our method uses a Fourier method. The approximate Newton iteration for the CH equation is

$$\left(1 + k\varepsilon^2\Delta^2 - kC\Delta\right) u^{r+1} = k\Delta\left(u^r\right)^3 + u^n - k\Delta u^n. \tag{2.32}$$

We will prove that (2.32) is a contraction mapping for small enough k. We first change (2.32) into a system of equations,

$$\begin{cases} U - kC\Delta U - k\Delta P = v \\ P + \varepsilon^2\Delta U = u^3 - v \end{cases}, \tag{2.33}$$

where $v = u^n$. For the partition $\bigcup_{k_h}$ in the finite element space $\sigma_h = H_h^1(\Omega)$, we look for a pair $(U, P) \in \sigma_h$ such that

$$\begin{cases} \langle U, \phi\rangle + kC\langle \nabla U, \nabla\phi\rangle + k\langle \nabla P, \nabla\phi\rangle = \langle v, \phi\rangle \\ \langle P, \psi\rangle - \varepsilon^2\langle \nabla U, \nabla\psi\rangle = \langle u^3 - v, \psi\rangle \end{cases} \tag{2.34}$$

holds for any $\phi, \psi \in \sigma_h$. We define a mapping $\Pi_v : \sigma_h \to \sigma_h$ for any $v \in \sigma_h$, so that $U = \Pi_v(u)$.

Lemma 2.5.1. *For given v with $\|v\|_2 = \alpha$, there exists a constant $\beta > 0$ such that Π_v*

maps the ball $S = \{u \in \sigma_h : \|u\|_2 \leq 2\alpha\}$ *into itself if* $k \leq \beta\varepsilon^2 h_\delta$. h_δ *is the mesh size related parameter.*

Proof. Letting $\phi = \varepsilon^2 U$ and $\psi = kP$ in (2.34), we have

$$\begin{cases} \varepsilon^2\|U\|_2^2 + kC\varepsilon^2\|\nabla U\|_2^2 + k\varepsilon^2\langle\nabla P, \nabla U\rangle = \varepsilon^2\langle v, U\rangle \\ k\|P\|_2^2 - k\varepsilon^2\langle\nabla U, \nabla P\rangle = k\left\langle u^3 - v, P\right\rangle \end{cases}. \tag{2.35}$$

Summing the two equations gives

$$\varepsilon^2\|U\|_2^2 + kC\varepsilon^2\|\nabla U\|_2^2 + k\|P\|_2^2 = \varepsilon^2\langle v, U\rangle + k\left\langle u^3 - v, P\right\rangle, \tag{2.36}$$

and applying the identity $a^2 - ab = \frac{1}{2}a^2 - \frac{1}{2}b^2 + \frac{1}{2}(a-b)^2$ leaves us with the inequality

$$\|U\|_2^2 + 2kC\varepsilon^2\|\nabla U\|_2^2 + \frac{2k}{\varepsilon^2}\|P\|_2^2 \leq \|v\|_2^2 + \frac{2k}{\varepsilon^2}\left\langle u^3 - v, P\right\rangle. \tag{2.37}$$

Using Young's inequality, it follows that

$$\|U\|_2^2 + 2kC\varepsilon^2\|\nabla U\|_2^2 \leq \|v\|_2^2 + \frac{k}{\varepsilon^2}\|u^3 - v\|_2^2. \tag{2.38}$$

By inverse estimates in the finite element space, there exists $\tilde{C} > 0$ independent of h_δ and α such that $\|u\|_\infty^2 \leq \tilde{C}\alpha h_\delta^{-\frac{1}{2}}$ for $\|u\|_2 \leq 2\alpha$, where h_δ only depends on mesh size. $\tilde{C}$ only depends on the degree of the underlying polynomial, so we can choose $C = \left(\tilde{C}\alpha h_\delta^{-\frac{1}{2}}\right)^2$ or

more practically $\|u\|_\infty^2$ so that

$$\begin{aligned}\|U\|_2^2 + 2kC\varepsilon^2\|\nabla U\|_2^2 &\leq \|v\|_2^2 + \frac{2kC^2}{\varepsilon^2}\|u\|_2^2 + \frac{2k}{\varepsilon^2}\|v\|_2^2 \\ &\leq \alpha^2 + \frac{8kC^2\alpha^2}{\varepsilon^2} + \frac{2k\alpha^2}{\varepsilon^2}. \end{aligned} \tag{2.39}$$

Thus it suffices to require that $8kC^2 + 2k \leq 3\varepsilon^2$ or equivalently $k \leq \frac{3\varepsilon^2}{8C^2+2} \leq \beta\varepsilon^2 h_\delta$. □

Similarly, we can show that Π_v is a contractive mapping if $k \leq \beta\varepsilon^2 h_\delta$ for some $\beta > 0$, but the proof is nearly identical so we will not repeat it here.

2.6 CPU version of code

We found solutions to the Functionalized Cahn-Hilliard equation by applying a spectral method to (2.5) with F_c and F_e given by (2.12) and (2.13) respectively. We wrote the computer code in C++, but any number of languages could have been used. There are two primary reasons we chose a spectral method; thin interfaces between phases and simplicity of taking numerical derivatives.

First, spectral methods offer the highest possible spatial resolution. Due to the physics of the materials modeled, solutions to the FCH equation develop $O(1)$ changes over $O(\varepsilon)$ intervals in space, so high spatial resolution is critical. We find that we can obtain reasonable solutions when we have as few as three grid points over the length of ε. For a cubic domain with 128 grid points along each edge, this resolution allows us to effectively capture up to forty-two changes in phase from polymer to water or visa versa, which is far more than we need in any reasonable simulation.

Second, the properties of a spectral method simplify derivatives in higher dimensions. The

number of Laplacians needed to calculate the solution at each time step becomes difficult with other methods, but in the frequency domain, each Laplacian is a simple element-wise multiplication of the Laplacian operator array against the target array. The spectral method brings with it some other challenges. First and foremost is the computational cost of transforming large arrays back and forth between the frequency and spatial domains. Fortunately, the FFTW package provides the tools to do Fourier transforms quickly and can easily be adapted to parallel computing on multiple CPUs.

Another restriction that comes with spectral methods is limitations on the domain shape and boundary conditions. Domain shape is limited to rectangular prisms since grid points must be spaced regularly in every direction. Further, the number of grid points in every direction should be a power of two to use the full speed of FFTW. When computing solutions spectrally, periodic boundary conditions come automatically, but any other boundary conditions become difficult or impossible to compute. Fortunately for us, periodic boundary conditions are reasonable for our simulations, but in future work, we may need to apply Dirichlet, Neumann, or mixed boundary conditions. At that time it will probably be necessary to switch methods.

2.7 Adaptive Time-Step

In addition to the advantages of a spectral method, we can speed up the simulation by adapting the time step to the evolution of the solution. The balance between Cahn-Hilliard like coarsening and surface interface generation causes periods of rapid change in the solution interspersed between slow evolution. Due to the unconditionally stable nature of the numerical scheme, we are free to choose a time step as large as we would like at the cost of

numerical error. When choosing a time step size, we estimate the evolution with a completely explicit calculation based on the previous time step size. The residual is

$$R^n = k\left[F_c\left(U^n\right) - F_e\left(U^n\right)\right] \approx U^{n+1} - U^n. \tag{2.40}$$

Then we determine the new time step

$$k_{\text{new}} = \frac{c_t}{\max\limits_{x\in\Omega}\left(R^n\right)}, \tag{2.41}$$

where c_t is a constant that allows us to control the time step sizes globally and is typically a value near 1. In future work, we intend to replace this crude estimate with a control on the error rather than the change in the solution.

A simulation that captures the importance of adaptive time stepping for the FCH is that of a sphere with an $O\left(\varepsilon^2\right)$ perturbation. A sphere that has too large of a radius is an unstable steady state of the equation, so with a small perturbation the solution has very little change for a long time period. Eventually the evolution moves the solution away from the unstable steady state and collapses into a lower energy geometric solution. A graph of the energy that shows the necessity of adaptive time stepping is in Figure 2.1.

It is easy to see the value of a time step that adapts to the solution. For evolution before $T = 600$, large time steps can be taken because there is very little change, but as the solution begins to change rapidly very small time steps must be taken in order to preserve the integrity of the approximate solution. Figure 2.2 gives still shots of the solution at important time intervals. Comparing to the energy, we see that the solution changes very little between $T = 1$ and $T = 600$ and it would be a severe waste of computing time if we

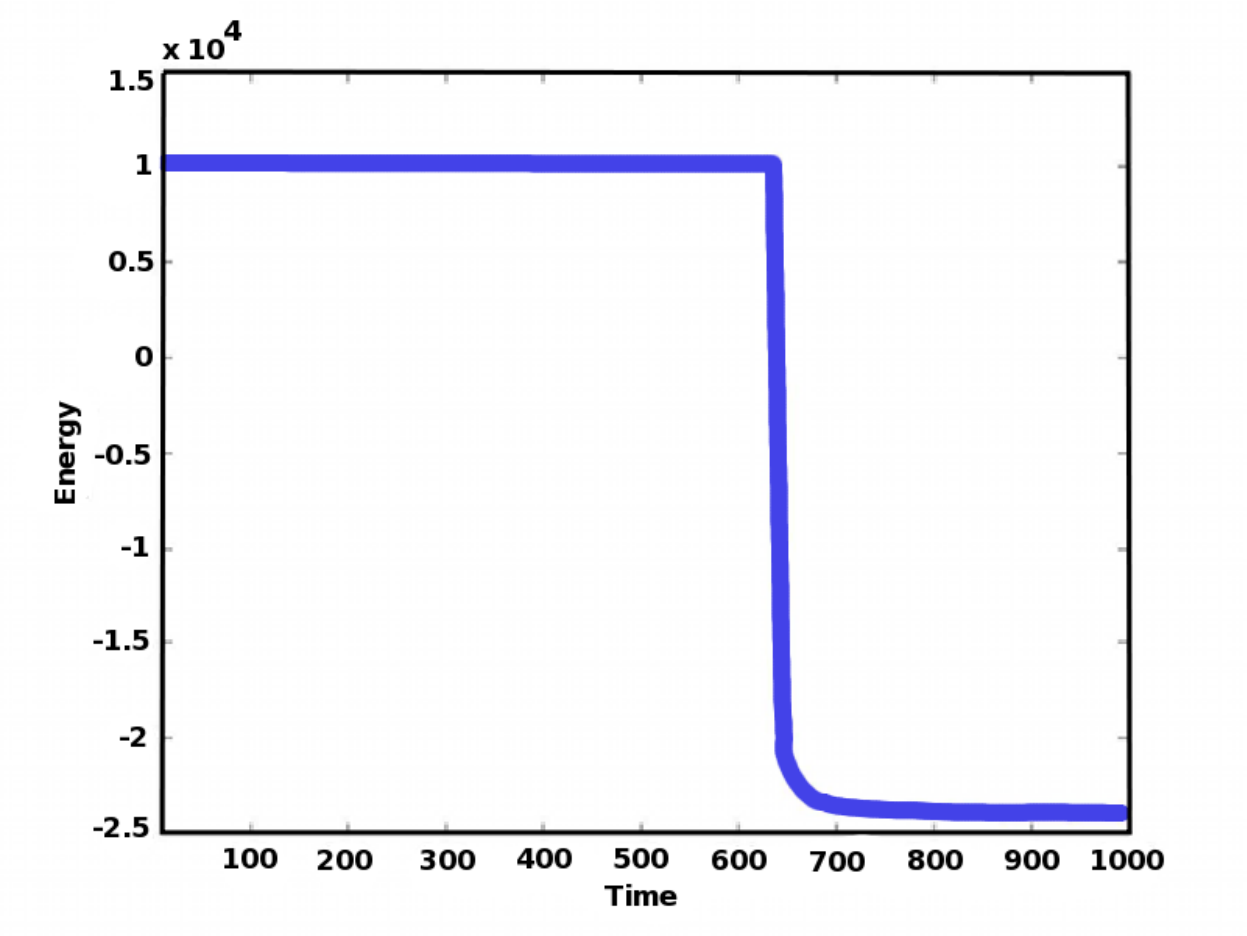


Figure 2.1 Energy trace of solution with unstable equilibrium initial condition.

used uniform time steps that were small enough to capture the changes at $T = 630$ and later. After $T = 700$ the evolution of the solution slows, and it is reasonable to take larger time steps again.

2.8 FCH on Graphics Processing Units

A recent development in scientific computing is General-Purpose computing on Graphics Processing Units (GPGPU). In the 1990s, Graphics Processing Units (GPUs) were developed to accelerate the building of 2D and 3D images for output to a display. Since then, the stream processing capabilities of GPUs have been turned to scientific computing, and the

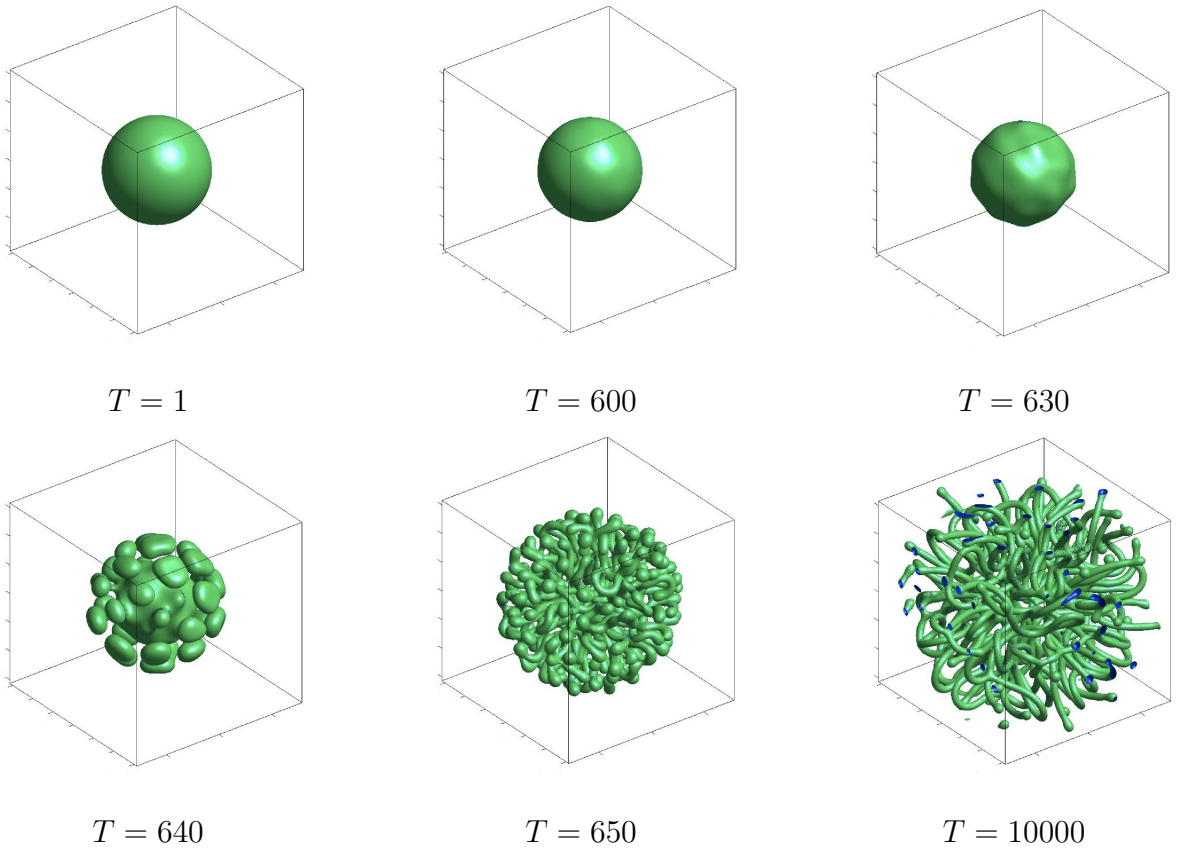


Figure 2.2 An unstable steady state solution for the FCH equation

application to scientific computing has lead to commercial lines of GPUs that have hundreds of computation cores but no display output hardware. GPU computation can be much faster if a problem or computation lends itself well to many lightweight computation threads. We used Nvidia GPU cards and the CUDA programming language which is C++ with extra functions to control the device (GPU) from the host (CPU).

When simulating the FCH equation spectrally with the gradient splitting detailed in section 2.3, there are three main types of mathematical operations; Fourier transforms (FFT), inverse Fourier transforms (IFFT), and multiply/add calculations. All three of these operations can be adapted very well to GPU processing. Three dimensional Fourier transforms can be computed extremely fast on a GPU by breaking the domain into lines of grid points,

then handing each line of grid points to a separate processor for calculation of the one dimensional FFT. After each processor is done computing the frequency representation of the data, the GPU can recompile the data back together. Inverse Fourier transforms can be easily computed for the same reason. The cuFFT package has already been developed to implement FFT calculations on the GPU, and it's easy to implement because the syntax and structure matches almost exactly with the FFTW library for C++.

Once the data can be quickly converted into either the spatial or frequency domain, the entire equation is simply element-wise multiplication or addition in the appropriate domain. This plays directly into the strength of the GPU, millions of lightweight computation threads. There are however limitations to GPU computing that cannot be overlooked. Unlike a CPU that has access to any memory location, GPUs multiprocessors can only access the data that is stored in the GPU card's memory. This leads to two complications; limitations on array size and movement of data. The GPU card we used had only 4GB of memory on the card, so for the number of double-precision, complex-valued arrays we used in the simulation, the size of the domain was limited to 2^{23} total grid points. For a cubic domain, this limit corresponds to 128 grid points in each dimension. It is important to realize that this is a rigid memory limit because it is defined by the GPU card's hardware.

The second complication is moving data to the memory on the GPU. The movement of data on and off the card is the only way to access the solutions to the PDE, and data movement is much slower than computation speed. Moving arrays onto the card for every calculation would erase all of the speed-up obtained from using the hundreds of processors. Our answer to this limitation was to put the initial condition array on the card at the beginning and not take the array off the card except when the simulation ended or we needed to record the data. Without this adjustment when porting the code from C++ to

CUDA, the change would have been pointless.

The final limitation that needs mentioning is the limit on double precision computations. There are some GPUs that are not capable of performing double precision calculations, but on the GPU we used, there was one double precision core on each multiprocessor, as opposed to eight single precision cores. This made the simulation much slower when we used double precision, but recent advances in GPU technology have rectified this. The newest GPUs for scientific computation have only double precision cores, which gives them up to sixteen times more double precision capability [80].

2.9 GPU Speed-Up

We conducted a speed test to compare simulation code written for parallel CPUs against the code written for the GPU. The CPU code was written in C++ and fully parallelized using OpenMP. The parallel version of FFTW was used for the Fourier transforms. The code was run on two Intel E5530 processors with four cores per processor. The processors have a clock speed of 2.4 GHz. The GPU used was a Tesla M1060 with 240 multiprocessor cores with a clock speed of 1.3 GHz, and it had 4GB of global memory. The two hardware configurations were both about two years old at the time of the test, so the speed result are comparable. The test was performed on initial conditions of random initial data for time up to $T = 0.001$, which required approximately 20 time steps. The solutions given by the two codes were exact up to implicit iteration precision. The simulation length was relatively short compared to our typical FCH simulations that compute solutions up to $T = 10,000$ or even $T = 100,000$. This was to show the impact of slow transfer of data between the host and GPU device.

Figure 2.3 gives the speed-up comparison for single and double precision calculations.

For both the single and double precision experiments, the speed-up for a single time step was much higher than the total time speed-up due to the slow transfer of initial data onto the GPU card at the beginning of the simulation and transfer of the final solution off the card at the end of the simulation. As the simulation time grows longer, the total time speed-up converges to the speed-up per time step value since the data transfer to and from the GPU will become insignificant. The jagged nature of the single precision, per-time-step curve is due to the error in measuring extremely small computation times. If the same speed-up experiment was run on a state of the art GPU and parallel CPUs, the results are expected to be similar, however the improvement in number of double precision cores per multiprocessor on the Fermi line of GPUs leads us to expect results more like the $\sim$ 25x speed-up of the single precision experiment [80].

2.10 Comparison of results to experimental data

A mathematical model is only as good as its ability to describe real materials. Unfortunately, three dimensional images of Nafion are not available, but there are systems composed of amphiphilic diblock copolymer systems that obey similar surface energy dominated physics. Figure 2.4 shows a comparison between a previous 2D simulation on the FCH equation [28], and images from an amphiphilic diblock copolymer system experiment [81]. Three dimensional simulations also show pore network solutions that self-assemble from random initial data. Figure 2.5 shows a three dimensional pore network along with two other solutions that all had identical random initial data. The difference between the geometries is small changes in the parameters η_1 and η_2 in equation (1.10).

Another geometry that has been seen extensively in experiments is the pearled pore. Figure 2.6 shows an image from an experiment performed by Bendejacq et al. on a diblock copolymer system [82]. Next to the experimental image is a simulation of pearling from the FCH equation. The numerical simulation had initial conditions of a straight pore with radially symmetric $O(\varepsilon)$ noise. The periodic boundary conditions prevent the pore from extending or contracting, and thereby force the solution into a frustrated quasi-steady state. We have not yet been able to find steady state pearling from random initial data in three dimensions, but we hope to find pearled pores and other interesting structures with a future parameter scan.

2.11 Conclusions

We extended Eyre's convex splitting scheme to the Functionalized Cahn-Hilliard equation and obtained a numerical scheme. We proved that the scheme had only one solution and that the energy could not increase for any size time-step. This shows that the method is unconditionally gradient stable with respect to the size of the time-step taken. The right hand side of the equation could not be split additively as desired, but by moving the mixed term into the implicit portion, we were able to obtain an effectively gradient stable method.

A novel iterative technique significantly increased the calculation speed by eliminating the need to rebuild the inverted operator for each iteration. The operator is built once at the beginning of the computation saving computation time. Using this temporal scheme with a Fourier spectral method for the spatial discretization made a scheme that could be implemented on a graphics processing unit. The GPU code is extremely efficient, and the speed up allowed us to study large, three-dimensional domains over long evolution times.

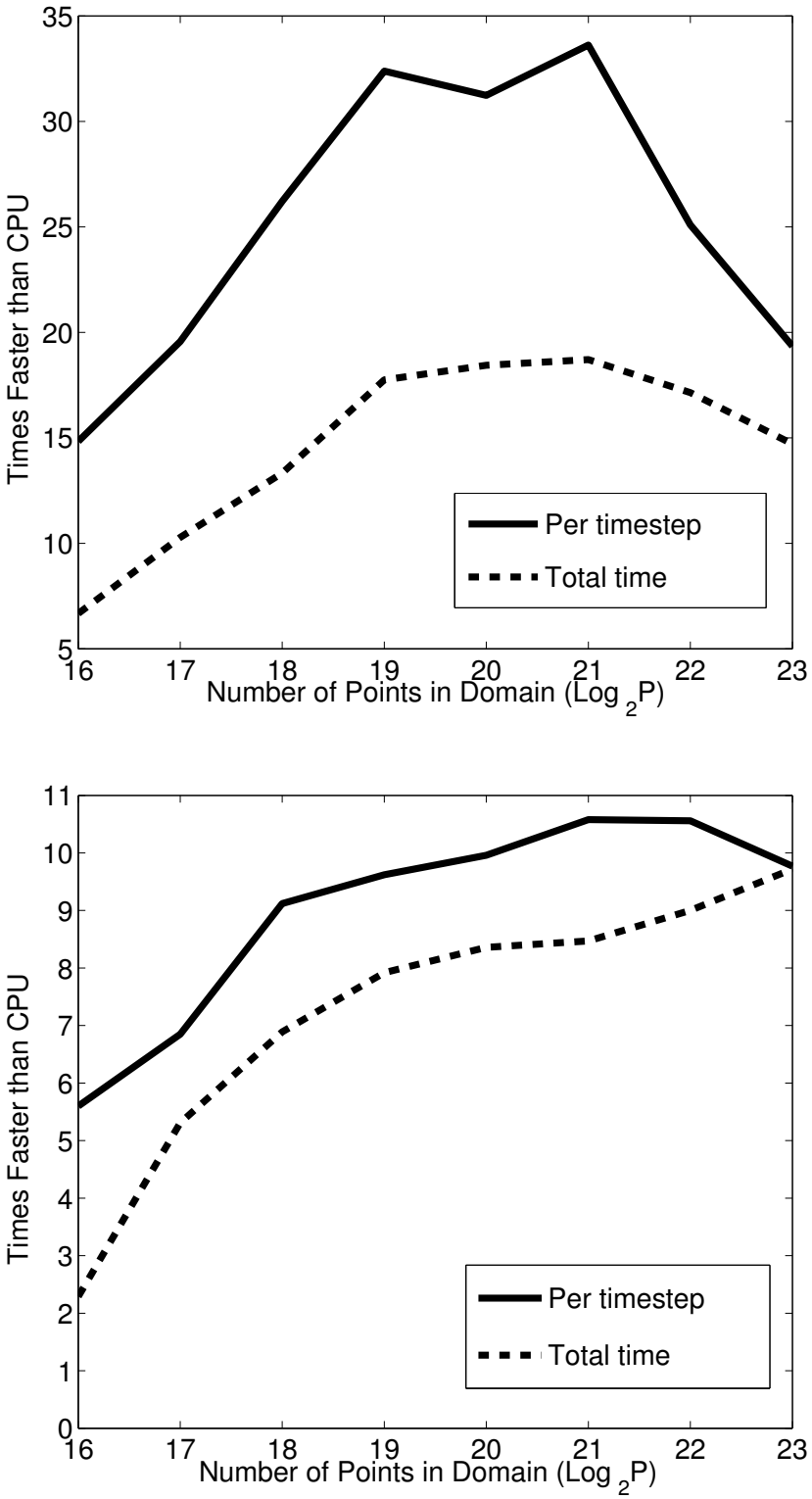


Figure 2.3 Speedup of GPU over parallel CPU for a short FCH simulation computed using single precision (top) and double precision (bottom).

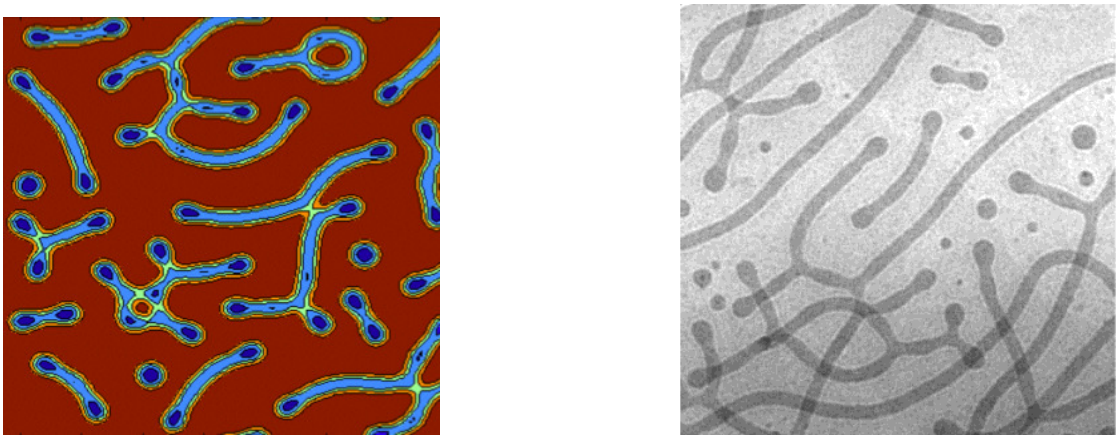
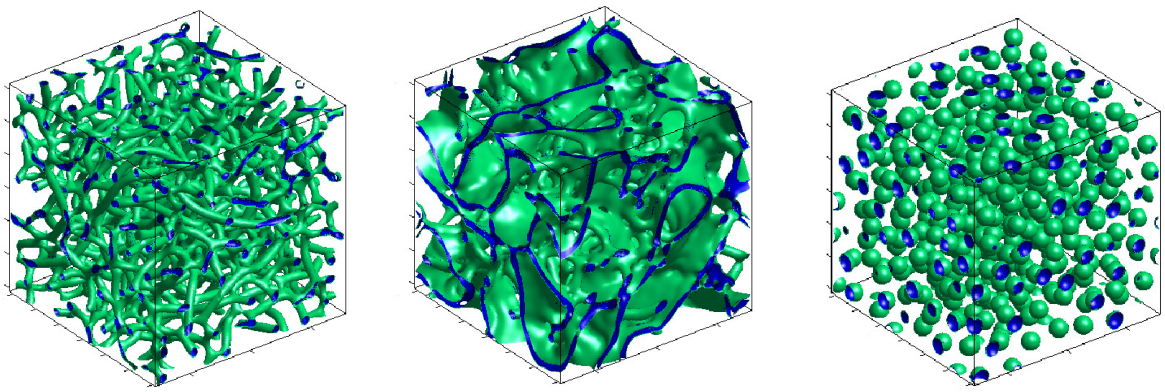
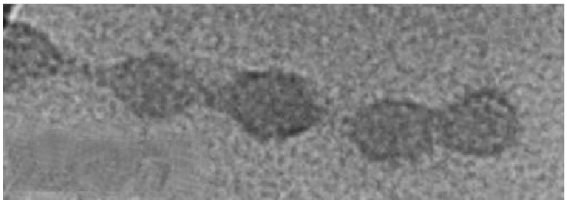


Figure 2.4 Two dimensional FCH simulation results (left) compared against images from a diblock copolymer experiment [81] (right)

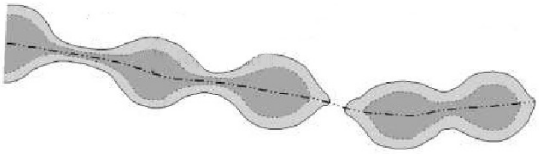


Pore network Lamella network Inverted micelles

Figure 2.5 Geometries that minimize the FCH energy for slightly different values of η_1 and η_2. The images show a level set near $u = 0$ with blue on the water side of the level set and green on the polymer side



TEM experimental image



Experimental diagram

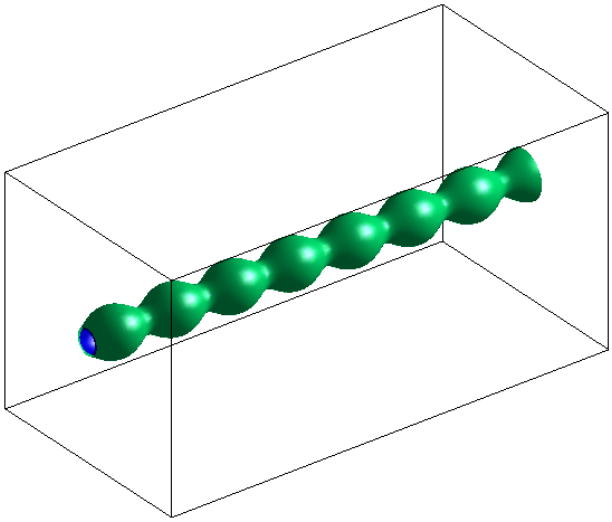


Figure 2.6 Experimental results for a diblock copolymer pore showing pearling instability [82] (left) compared against result of a numerical solution of the FCH equation (right)

Chapter 3

Fully Implicit Method

3.1 Introduction

Many material science problems require an understanding of the microstructure that develops in a mixture of two of more materials or phases over time as it phase separates during a casting or annealing process. One such equation is the well-studied Cahn-Hilliard [10] equation, written below in equation (3.2) in a one-dimensional (1D) setting, that describes a binary alloy during annealing. The parameter ϵ in the model describes the width of the layers between the regions. Such regions form in $O(1)$ time in a *spinodal* evolution. Subsequently, they merge in a *ripening* process. Ripening happens on longer time scales, generically $O(e^{C/\epsilon})$ for 1D Cahn-Hilliard [83] and $O(1/\epsilon)$ in higher dimensions [11]. We extend the use of the terms "spinodal" and "ripening" to describe similar regimes in the evolution described by other equations. Phase regions undergoing Cahn-Hilliard evolution increase in size over time in a coarsening process. The statistics of this coarsening process are of interest [84].

The Cahn-Hilliard model is a sub-class of phase field models. A review of the use of such models in material science applications can be found in [6]. It can be shown rigorously that as $\epsilon \to 0$, solutions of Cahn-Hilliard equations have layers that tend to interfaces that move with a nonlocal geometric motion known as the Mullins-Sekerka flow [11]. Other phase field models also limit to geometric motion of other kinds. Understanding the limiting process

and studying it directly is of interest. In addition, Cahn-Hilliard equations and variants can be used in computational approximation of moving interfaces in so-called *diffuse interface methods* [85, 86] in which the problem for u is coupled to other variables describing other physics. While the computational approach developed in this chapter might be useful to some diffuse interface computations, we are motivated by a general class of pure (uncoupled to other physics) energy gradient phase field problems described below.

There are several interesting generalizations of the Cahn-Hilliard equation. A lower order version (two instead of four spatial derivatives), the Allen-Cahn equation (3.1) [87] is also of interest in materials science, describing the evolution of crystal grains of the same material during annealing. This equation can also be called a Ginzberg-Landau equation.

The aim of this chapter is to develop a numerical approach that can be applied to a wide range of phase field problems that can easily be adapted to new terms, higher order problems, and extension to vector solutions. It should be made clear we do not attempt to outperform well-developed codes with space and time adaptivity with fast, multi-grid solvers that have been developed for particular problems. Rather, we develop a reasonably fast time-adaptive technique with general applicability.

Since many questions of interest in materials science are about the microstructure of a bulk material far from boundaries, it is reasonable to consider problems in periodic domains. We use a Fourier spectral discretization which is a natural choice in this setting. Although this does rule out spatial adaptivity, it does admit a fast implementation on Graphical Processing Units (GPU) in the computational framework we develop. We discretize in time using Backward Differentiation Formula (BDF) methods [88] of low order, which have good stability properties. Temporal error estimation is done with Adams-Bashforth (AB) [89] predictors. Newton's method is used to solve the resulting nonlinear problems. The Jacobian

matrix in the solve for the Newton update is symmetric since it is the second variational derivative of an energy functional. It is also positive definite for time steps small enough (this is discussed in more detail below). Although the Jacobian is dense for spectral discretizations, multiplication by the matrix can be done quickly using the Fast Fourier Transform (FFT). This motivates our use of the conjugate gradient method [90] to solve the Newton updates. Such an approach used on high order problems requires an efficient preconditioner. We use a constant coefficient version of the problem that is a linearization at pure phase states, which will dominate the solution during ripening. This idea is similar to that used in [91] in fixed point iterations for time stepping for Cahn-Hilliard with operator splitting. Efficient performance is seen with our approach for a wide selection of scalar and vector problems from second to sixth order. Mild increase is seen in preconditioned conjugate gradient (PCG) iteration counts per time step as the time step is increased and ϵ is decreased. Exploration of the performance of the method specifically for the 1D Cahn-Hilliard problem, which has a well understood structure during ripening, shows that the number of PCG iterations per solve scales as $O(\sqrt{\delta t/\epsilon})$ for large time steps δt and small ϵ, independent of the spatial discretization.

There have been many contributions to the numerical solution of the Cahn-Hilliard and related equations. Our work is novel in four ways: we exploit the symmetry of the Jacobian matrix for the fully implicit time stepping problem in a CG method; we propose and analyze the preconditioner for this Jacobian solve and show that it is effective for a number of problems; we implement the method in modern GPU architecture and get fast performance; we demonstrate that the Jacobian matrix is not singular for large time steps during ripening and that fully implicit time stepping leads to accurate solutions with these large time steps with energy decrease.

We discuss further the issue of time-stepping. For arbitrary discrete initial data, the fully implicit time discretization problem is known to have a unique solution only when the time step is small enough [39]. As the spatial grid size $h = \Delta x$ is reduced and the initial data on the refined grid is again allowed to be arbitrary, the time step that guarantees unique solutions to the implicit time discretization problem is reduced. In addition, it is not possible in general to show that a fully implicit time step leads to a decrease in the underlying energy. Guarantees of solvability for any time step δt and energy decrease are possible for some models with an operator splitting approach due to Eyre [46]. Although never published, this work has been very influential and some of the results are summarized in [49]. Several of the computational approaches cited in Chapter 1 use variants of this splitting approach. Some of the splitting techniques lead to nonlinear problems and we show that our ideas lead to efficient solution of these problems (with preconditioned CG iteration counts independent of δt and ϵ).

Guarantees of energy decay and unique solutions for any chosen time step δt are very attractive and so splitting techniques have dominated the thinking in numerical methods for these problems for many years. Our use of fully implicit time stepping may be seen as controversial. We first considered this approach because of some motivating high order, vector models for which the process of splitting the gradient terms into convex and concave parts was not straightforward. When implemented, we discovered that fully implicit time stepping did not suffer from severe time step restrictions as the literature predicted. That previous analysis was really for a "worst case" scenario and the solution structure through ripening processes allowed for large time steps to be taken. We also discovered computationally that Eyre's splitting can lead to disproportionately large temporal errors during ripening and prohibits the use of large time steps appropriate to the dynamics there if time-accurate

solutions are required. This poor behavior can be understood with formal asymptotics on a simple problem shown below. Our fully implicit time stepping is able to take appropriate large time steps and maintain accuracy. While not the original intention of our work, our results suggest that fully implicit time stepping strategies are more efficient than popular splitting techniques when time accuracy is desired, at least for the selection of models we consider.

We outline the following sections of the paper. In Section 3.2 we present the equations for the various models we consider and how they arise from an energy gradient flow. In Section 3.3 we give a basic description of the approach in a 1D setting using simple backward Euler time stepping. This is done for clarity of exposition, but it should be made clear that the approach has wide applicability, shown in Section 3.5 for a number of models in 2D and in Section 3.6 where a GPU implementation to Cahn-Hilliard in 3D is described. Higher order time stepping methods are discussed in Section 3.7. The performance of the preconditioner is examined numerically and with formal asymptotics in a simple 1D setting in section 3.4. A similar approach is taken in section 3.9 to investigate time stepping with operator splitting.

3.2 Models

We consider first the very basic model for a scalar function $u(x,t)$ in a 1D setting of the Allen-Cahn [87] equation

$$u_t = \epsilon^2 u_{xx} - W'(u) \tag{3.1}$$

and Cahn-Hilliard [10] equation

$$u_t = -\epsilon^2 u_{xxxx} + (W'(u))_{xx} \tag{3.2}$$

where $W(u) = \frac{1}{4}(u^2 - 1)^2$. This is simply the one-dimensional version of 1.3. We consider $x \in [0, 2\pi]$ and u to be periodic in space.

Qualitatively, the reaction terms $W'(u)$ drive the solution to the two pure states $u = \pm 1$ and the other terms smooth the interface between regions of different phases over a width of $O(\epsilon)$. These equations can be written with the ϵ in different places corresponding to different time scalings. It is important to note that the results discussed below on the condition number of the preconditioned conjugate gradient method and its dependence on time step δt and ϵ are with respect to the scaling shown in equations (3.1) and (3.2) above. Higher dimensional versions of these equations are obtained by replacing $\partial^2/\partial x^2$ by the Laplacian Δ.

The models above are gradient flows on the Cahn-Hilliard energy defined in 1.1. The fact that the evolution is a gradient flow leads to a symmetric Jacobian matrix for the implicit time step of the discretization, allowing the use of the conjugate gradient method for its solution. Note that the Allen-Cahn model is a gradient flow in the standard L_2 inner product:

$$(u, v) := \int_0^{2\pi} uv dx. \tag{3.3}$$

However, the Cahn-Hilliard model is a gradient flow with respect to the H_{-1} inner product:

$$(u, v)_{H_{-1}} := (u, \Delta^{-1} v). \tag{3.4}$$

In addition, we consider the scalar sixth order problem 1.10 where η is a given *positive* constant (note that with this sign, this term promotes the formation of phase interface). This is also a gradient flow of a certain energy in the H_{-1} inner product. Models of this form are of current interest in the study of pore formation in functionalized polymers [92].

We also consider the following vector model for $\mathbf{u} = (u, v)$:

$$\mathbf{u}_t = -\epsilon^2 \Delta\Delta\mathbf{u} + \Delta\nabla_{\mathbf{u}} W(\mathbf{u}) \tag{3.5}$$

where here

$$W(\mathbf{u}) = \prod_{i=1}^{3} |\mathbf{u} - \mathbf{u}_i|^2 \tag{3.6}$$

and $\mathbf{u}_i$ are the points in the (u, v) plane that correspond to the cube roots of unity. This is a volume preserving model that forms symmetric triple junctions between three phases. It can be seen as the higher order mass preserving version of the Ginzberg-Landau equation presented in [93].

The extension of our computational method to higher dimensions and more complex (higher order, vector) models is relatively straight-forward. We consider this the main strength of our approach.

3.3 Basic numerical approach and results

3.3.1 Spectral discretization in space

In Section 1.9, we discretized the Cahn-Hilliard equation in space using a Fourier spectral method. The sixth order and vector models are discretized in a similar manner. It is a direct computation to show that the MOL discretizations above guarantee a decrease in a discrete energy:

$$\frac{d}{dt}\left\{ \frac{\epsilon^2 h}{2} |F^{-1}\Omega_\alpha F\mathbf{U}| + h\sum_{j=1}^{N} W(U_j) \right\} \leq 0$$

where in the left hand term $|\cdot|$ is the Euclidean norm and Ω_α is a diagonal matrix with entries $i\alpha_n$. The left hand term in braces is the discrete analogue of the energy (1.1). Since the MOL discretization has this property, we can expect time discretizations to have the property also, at least up to the order of the truncation error of the method. This is observed in the variety of computational examples shown in the rest of this work. In fact, with the adaptive time stepping strategy we use, an increase in this discrete energy is never observed in any accepted time step including very large time steps used during ripening events. We continue with a description of the time discretization of these models below.

3.3.2 Adaptive, implicit discretization in time

We consider the fully discrete approximation of u

$$u(jh, t_m) \approx U_j^m,\ j = 1, \dots N \text{ and } m = 0, \dots M$$

with $t_0 = 0$ where initial conditions are given and time steps $\delta t_m = t_m - t_{m-1}$ are chosen adaptively as shown below. A basic, first order approach is shown here. Higher order methods are presented in section 3.7. They provide some efficiency gains but are not overwhelmingly superior for modest accuracy.

Consider the semi-discrete Allen-Cahn equation (1.11). Starting at t_{m-1} we form the explicit, forward Euler predictor $\mathbf{U}^*$ for the solution at time t_m:

$$\mathbf{U}^* = \mathbf{U}^{m-1} + \delta t_m \left[\epsilon^2 \Delta_h \mathbf{U}^{m-1} - W'(\mathbf{U}^{m-1})\right]. \tag{3.7}$$

We use this as an initial step in a Newton iteration for the solution to an implicit, backward

Euler time step

$$\mathbf{G}(\mathbf{U}^m) := \mathbf{U}^m - \delta t_m \left[\epsilon^2 \Delta_h \mathbf{U}^m + W'(\mathbf{U}^m)\right] - \mathbf{U}^{m-1} = \mathbf{0} \tag{3.8}$$

We describe the solution procedure for this problem in the subsection below.

Expanding the predictor and corrector steps in Taylor series, it is easy to show that the exact local truncation error η_e for this step can be approximated by

$$\eta_e \approx \eta := \frac{1}{2}\|\mathbf{U}^* - \mathbf{U}^m\|$$

as is well known. We use maximum norms for all error calculations in this work. In time-adaptive computations below, we specify a given tolerance $\sigma > 0$ for the local truncation error. If a time step fails the accuracy check ($\eta > \sigma$) then we fail the step and repeat with the time step reduced by a factor of 1.3. This particular value is a somewhat arbitrary factor but there were very few failed steps in the computations shown in this work and changing this value does not change performance. A time step is also failed if the Newton iterations do not converge or if the step leads to an increase in energy (although as noted above, this failure was never observed in any of the computations described in this chapter). After a successful step, the next step is taken with time step

$$\delta t_{m+1} = \delta t_m \max\left(0.8\sqrt{\frac{\sigma}{\eta}}, 1.3\right)$$

where the local truncation error is assumed to be dominated by its leading order quadratic term and 0.8 and 1.3 are "safety factors".

3.3.3 Solution of the implicit system

Consider (3.8), the implicit problem to be solved at each time step for the discretization of the Allen-Cahn equation (3.1). Let $\mathbf{U}^{(r)}$ denote the r'th Newton iterate approximation of $\mathbf{U}^m$. The next iterate requires a solve with the Jacobian coefficient matrix

$$J = I - \delta t_m \epsilon^2 \Delta_h + \delta t_m \Lambda_2 \tag{3.9}$$

where Λ_2 is the diagonal matrix with entries

$$W''(U_j^{(r)}) = 3[U_j^{(r)}]^2 - 1.$$

The matrix J is dense but symmetric and multiplication by J can be done efficiently using the fast fourier transform and diagonal multiplication. This suggests the use of the conjugate gradient method as a solution technique. However, the condition number of J is large and the equivalent matrices for problems with higher order derivatives have even larger condition number. Efficient preconditioning is clearly required. We propose the preconditioner Q^{-1} where Q is the discretization of a constant coefficient problem

$$Q = I - \delta t_m \epsilon^2 \Delta_h + 2\delta t_m \tag{3.10}$$

which can be inverted efficiently. This is motivated heuristically by the observation that during ripening, the solution will have values approximately ± 1 at most grid points. The behavior of the preconditioned gradient solver is examined in computational studies below and analytically in section 3.4. For the discretization of the Allen-Cahn equation, it is shown

that the condition number is $O(\delta t)$ for large δt independent of ϵ during ripening.

For the discretization of the Cahn-Hilliard problem (3.2) we have the following Jacobian matrix J_{CH} and use the preconditioner Q_{CH}

$$\begin{aligned} J_{CH} &= I + \delta t_m \epsilon^2 \Delta_h \Delta_h - \delta t_m \Delta_h \Lambda_u \qquad (3.11) \\ Q_{CH} &= I + \delta t_m \epsilon^2 \Delta_h \Delta_h - 2\delta t_m \Delta_h. \qquad (3.12) \end{aligned}$$

Note that for this problem and the ones below, the matrices are symmetric in the discrete version of the H_{-1} inner product (3.4). In the preconditioned conjugate gradient (PCG) algorithm [90] these inner products are used. The preconditioned Jacobian matrix is shown to have condition number $O(\delta t/\epsilon)$ in ripening states in this case.

For the discretization of the sixth order problem (1.10) we have

$$\begin{aligned} J_6 &= I - \delta t_m \epsilon^4 \Delta_h \Delta_h (\Delta_h + \eta I) + \delta t_m \epsilon^2 \Delta_h \Delta_h \Lambda_2 \qquad (3.13) \\ &\quad + \delta t_m \epsilon^2 \Delta_h (\Lambda_L \Lambda_3 + \Lambda_2 \Delta_h + \eta \Lambda_2) \\ &\quad - \delta t_m \Delta_h (\Lambda_2^2 + \Lambda_1 \Lambda_3) \\ Q_6 &= I - \delta t_m \epsilon^4 \Delta_h \Delta_h \Delta_h - \delta t_m (\eta \epsilon^4 - 4\epsilon^2) \Delta_h \Delta_h \qquad (3.14) \\ &\quad - \delta t_m (4 - 2\epsilon^2 \eta) \Delta_h \end{aligned}$$

where here Λ_i, $i = 1, 2, 3$ is the diagonal matrix with entries

$$\frac{d^i W}{du^i}(U_j^{(r)})$$

and Λ_L is the diagonal matrix with entries $\Delta_h U^{(r)}$. Here, the preconditioner is derived with

the same heuristic reasoning as above, that at pure states $u = \pm 1$, $\Lambda_1 = 0$, $\Lambda_2 = 2I$, $\Lambda_3 = 6I$ and $\Lambda_L = 0$.

For the vector problem (3.5) we consider the solution components (U_j, V_j) at a grid point as a block and have

$$\begin{aligned} J_V &= I + \delta t_m \epsilon^2 \Delta_h \Delta_h - \delta t_m \Delta_h \Lambda_V \\ Q_V &= I + \delta t_m \epsilon^2 \Delta_h \Delta_h - 18 \delta t_m \Delta_h. \end{aligned}$$

where Λ_V is a block diagonal matrix with 2×2 blocks

$$\begin{bmatrix} \frac{\partial^2 W}{\partial u^2} & \frac{\partial^2 W}{\partial u v} \\ \frac{\partial^2 W}{\partial u v} & \frac{\partial^2 W}{\partial v^2} \end{bmatrix}$$

where the vector potential (3.6) is considered above and the partial derivatives are evaluated at the corresponding grid values $(U_j^{(r)}, V_j^{(r)})$. A straight-forward calculation shows that for values of $\mathbf{u}$ in any of the three symmetric potential wells, this block is diagonal with diagonal entries 18. Hence, the preconditioner follows the same reasoning as above.

3.3.4 Basic numerical results

We consider the 1D Cahn-Hilliard equation as the model system in this section. Starting from initial data

$$u_0(x) = \cos(2x) + \frac{1}{100} e^{\cos(x + 1/10)} \tag{3.15}$$

and $\epsilon = 0.18$, the system moves to an intermediate state at $t = 300$ with two intervals with $u \approx -1$ as shown in Figure 3.1. This time is roughly where the slowest ripening evolution

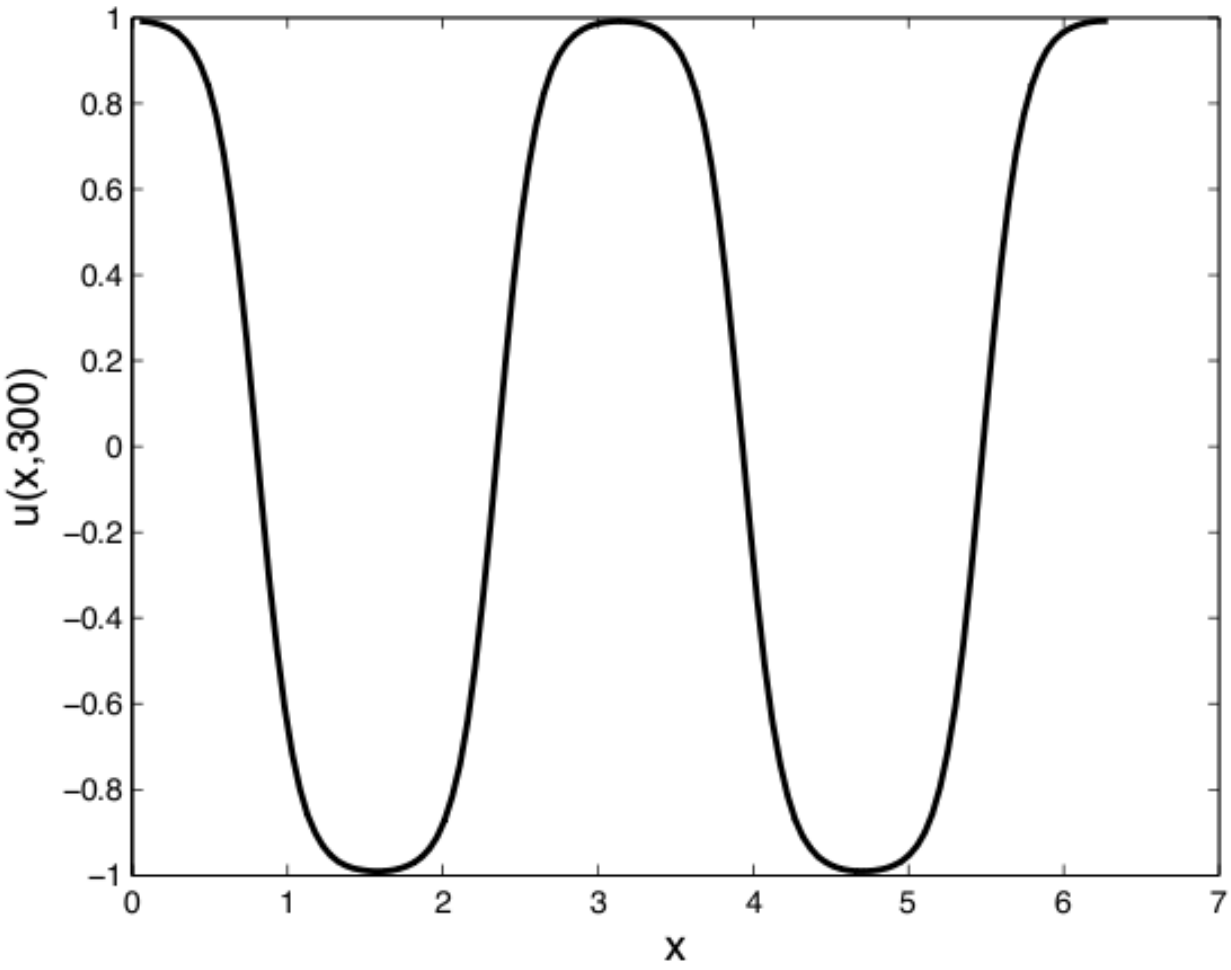


Figure 3.1 This figure corresponds to the solution of the 1D Cahn-Hilliard (3.2) with initial conditions (3.15) for $\epsilon = 0.18$ at time $t = 300$. This is the type of solution at which the preconditioner performance is examined in more detail in Section 3.4.

occurs, marked by the largest time steps taken by the method as shown in Figure 3.5. The second term on the right above is a small perturbation so that these two intervals are not symmetric, so that we will see generic behavior. At much longer times, these two intervals will slowly evolve and merge [11, 83] as shown below. The fixed time step δt performance of the method is examined at a short time $t = 0.2$ in Tables 3.1 and 3.2 . First order convergence in the time step and spectral convergence in N is seen as expected. Also observed is that $N = O(1/\epsilon)$ is needed to spatially resolve the interfaces of width ϵ.

δt	$E_{\delta t}$
2e-4	1.32e-5
1e-4	6.6e-6
5e-5	3.3e-6

Table 3.1 Error estimates $E_{\delta t} = \|\mathbf{U}_{\delta t} - \mathbf{U}_{\delta t/2}\|$ for fixed time step δt computations of the 1D Cahn-Hilliard model with $\epsilon = 0.18$ to short time $t = 0.2$. Spatial discretization is fixed at $N = 128$.

N	E_N for $\epsilon = 0.18$	E_N for $\epsilon = 0.09$
32	2.0e-3	0.139
64	9.3e-7	4.4e-3
128	9.0e-13	1.3e-6

Table 3.2 Error estimates $E_N = \|\mathbf{U}_N - \mathbf{U}_{2N}\|$ for computations of the 1D Cahn-Hilliard model with $\epsilon = 0.18$ and $\epsilon = 0.09$ to short time $t = 0.2$. Fixed time steps of $\delta t = 1e - 4$ are used.

The performance of the adaptive time stepping method through a ripening event is shown in Table 3.3. In these runs the Newton solve has residual tolerance 10^{-8} and the PCG solve at each step has tolerance 10^{-9}. Starting from initial data (3.15) the solution contains four transition layers at short time as shown in Figure 3.1. Over a very long time, the middle transitions move closer together and merge as shown in Figure 3.2. The final state with two transition layers is steady. The energy, time step size and PCG iteration history for the are shown in Figure 3.5 for the tolerance $\sigma = 10^{-4}$. One or two Newton iterations are taken at each time step. Note the sharp transitions in the energy $\mathcal{E}$ at early times (the spinodal evolution) and at the ripening event at which the time steps are small. The ripening time estimates shown in Table 3.3 correspond to the time t at which the midpoint value $u(\pi, t)$ changes from positive to negative. This ripening event happens at a very fast time scale after the long, slow transient. The results in Table 3.3 show that the method can accurately capture the time that such events occur. Since local truncation error is $O(\delta t^2)$ it is expected that as the local tolerance σ is reduced by a factor of 10, the number of total time steps

σ	time steps	ripening time	total PCG iterations
1e-4	848	8180	19,105
1e-5	2580 (3.04)	8273	39,942 (2.09)
1e-6	8072 (3.13)	8304	87,563 (2.19)
1e-7	25446 (3.15)	8314	227,799 (2.60)

Table 3.3 Performance of the adaptive time stepping through a ripening event of the 1D Cahn-Hilliard model with $\epsilon = 0.18$ and $N = 128$. The numbers in brackets are ratios to quantities in the previous row.

should increase by a factor of $\sqrt{10} \approx 3.16$ as is observed computationally. This validates our simple error estimation strategy. Note that the number of PCG steps increases by a smaller factor, indicating that the condition number of the implicit system decreases as $\delta t \to 0$. Note also that the performance of *the solver is independent of* N (for fixed ϵ).

It is well known [94, 83] that ripening is exponentially slow in ϵ in 1D Allen-Cahn and Cahn-Hillard models. With error tolerance $\sigma = 10^{-4}$ and $N = 128$ we compute ripening at time approximately 34,200 using 948 time steps and 22,950 PCG iterations for $\epsilon = 0.16$ and ripening time 218,000 using 1081 time steps and 28,417 PCG iterations for $\epsilon = 0.14$. This demonstrates that our approach behaves well when computing through ripening over very long time scales. It is known that to resolve the dynamics for increasing small ϵ, higher precision arithmetic is needed to resolve the exponentially small interactions of the layers in this 1D setting [83].

The method performs similarly well on the lower order Allen-Cahn equation (3.1). Spinodal evolution leads to similar layers to those shown in Figure 3.1. In this case the equations do not preserve mass and so ripening involves the separate collapse of the intervals of state $u \approx -1$ leading to a steady state of $u \equiv 1$.

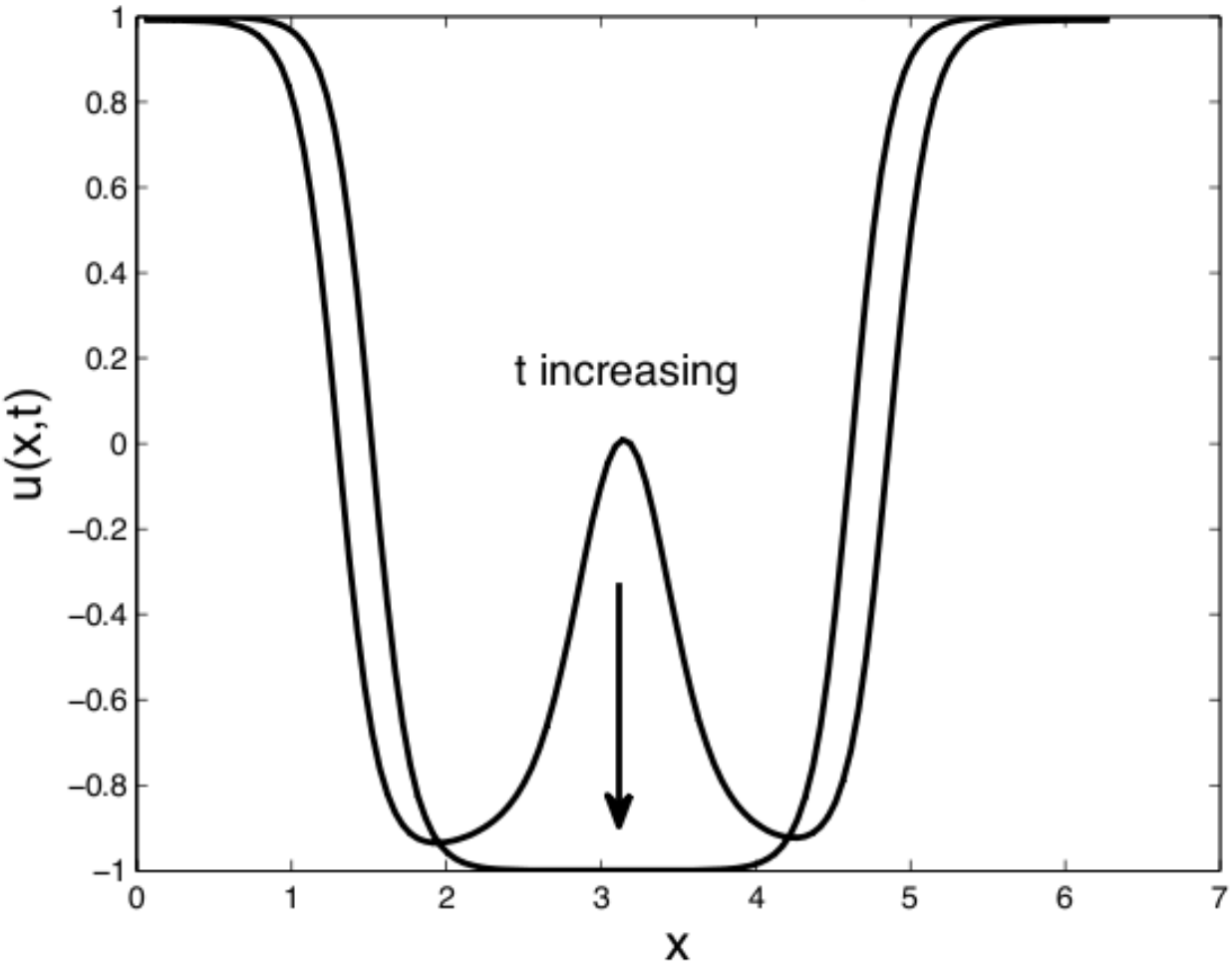


Figure 3.2 This figure corresponds to the solution of the 1D Cahn-Hilliard equation (3.2) with initial conditions (3.15) for $\epsilon = 0.18$ during the final stages of the ripening process at large time.

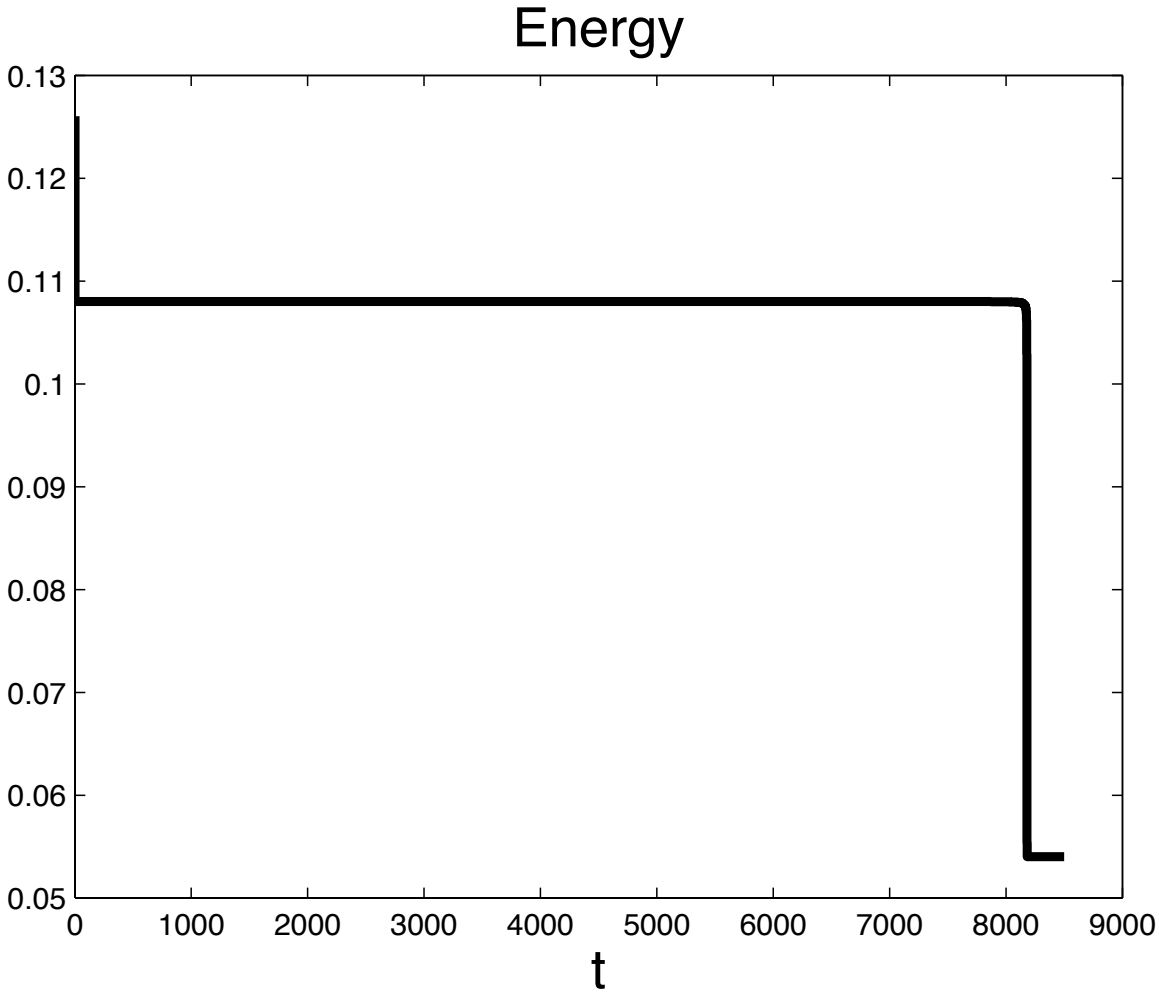


Figure 3.3 The energy history for the adaptive time approximation with tolerance $\sigma = 10^{-4}$ of the 1D Cahn-Hilliard equation (3.2) with initial conditions (3.15) for $\epsilon = 0.18$.

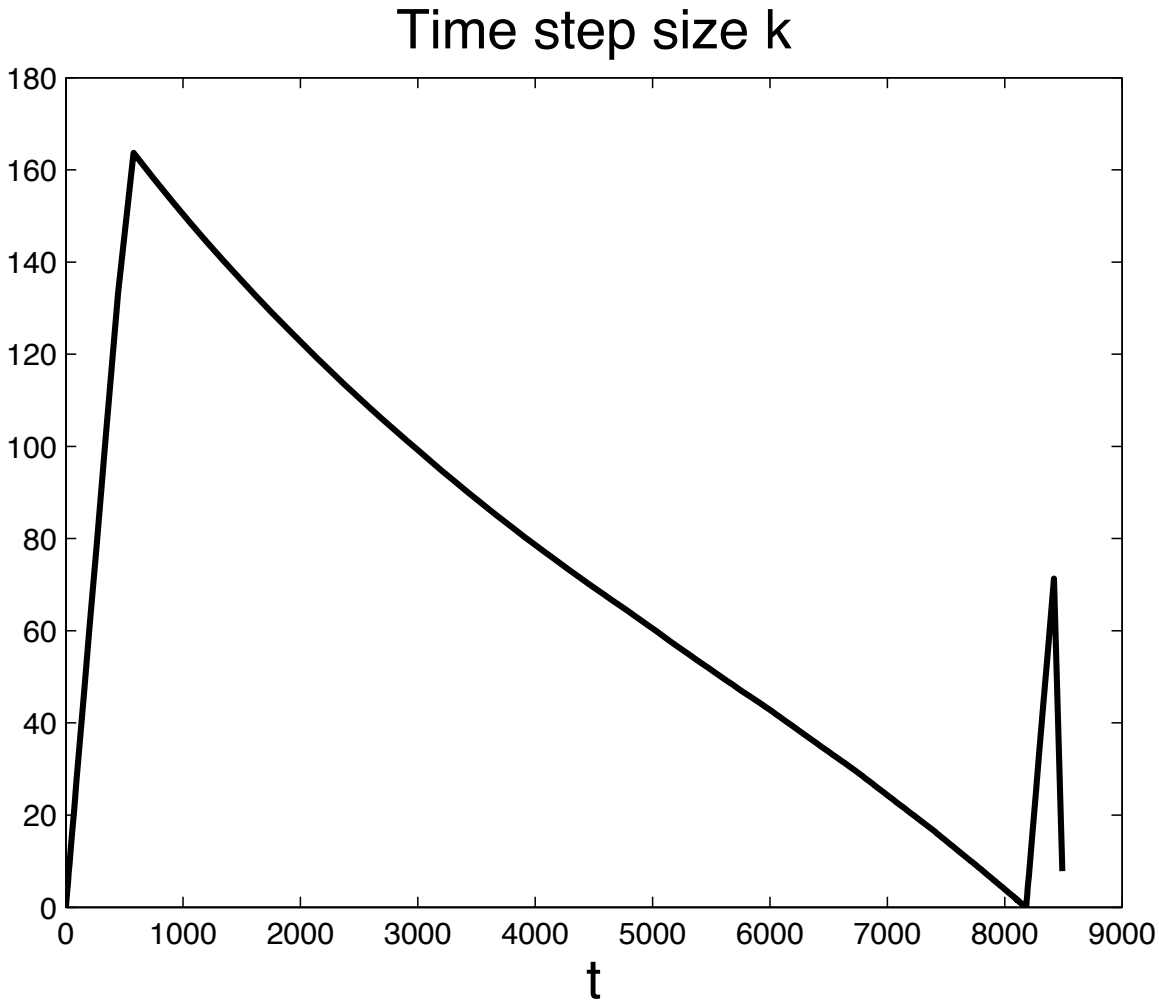


Figure 3.4 The time step history for the adaptive time approximation with tolerance $\sigma = 10^{-4}$ of the 1D Cahn-Hilliard equation (3.2) with initial conditions (3.15) for $\epsilon = 0.18$.

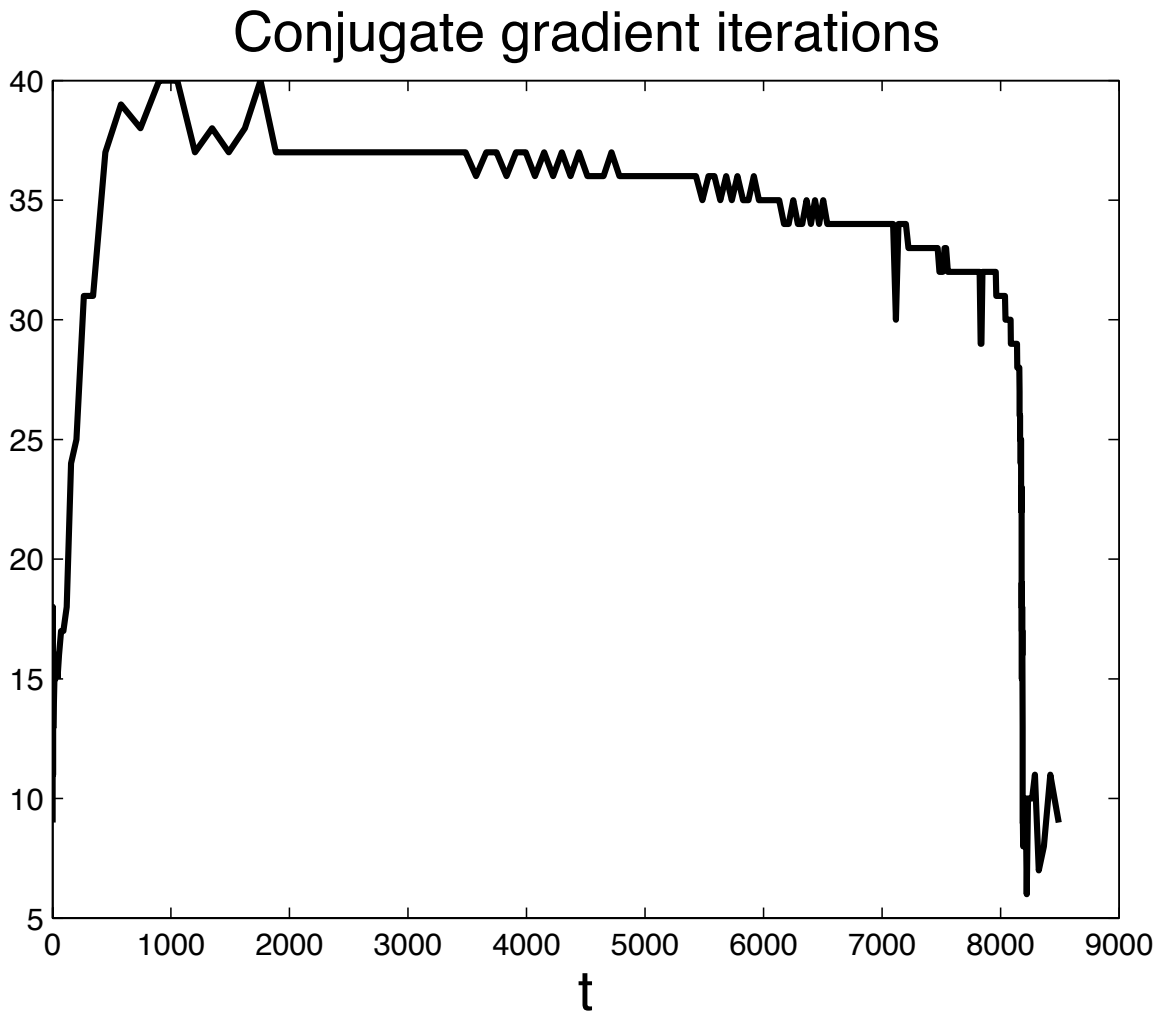


Figure 3.5 The PCG iteration count history for the adaptive time approximation with tolerance $\sigma = 10^{-4}$ of the 1D Cahn-Hilliard equation (3.2) with initial conditions (3.15) for $\epsilon = 0.18$.

3.4 Investigation of the preconditioned system

3.4.1 Preliminaries and numerical results

At a ripening state u of the 1D Cahn-Hilliard model evolving from the initial data (3.15), we examine the structure of the preconditioned Jacobian matrix

$$A = Q_{CH}^{-1} J_{CH}$$

as presented above. We can consider the operator A in the continuum limit. Since J_{CH} is a low order perturbation of the elliptic operator Q_{CH}, the limit operator A is a relatively compact perturbation of the identity (see [95], chapter 6). Thus we can expect the condition number of A to be relatively insensitive to the spatial discretization level N for N large enough to resolve the problem. In the discrete 1D setting, A and its eigenvalues can be computed explicitly using built-in MATLAB commands. The eigenvalues for the $\epsilon = 0.18$, $\delta t = 10$ at the solution shown in Figure 3.1 are shown in Figure 3.6. Note the clustering of eigenvalues near 1 as expected from the preconditioning. At this slowly evolving state, there are small eigenvalues as $\epsilon \to 0$ and $\delta t \to \infty$ that determine the condition number of A and limit the performance of the CG iterations. The eigenfunction corresponding to the smallest eigenvalue for parameters $\epsilon = 0.18$ and $\delta t = 10$ is shown in Figure 3.7. Notice that it is composed of pulses at the locations of the transition layers in the solution in Figure 3.1. For this state u there are $M = 4$ transition layers. It is shown using formal asymptotics in section 3.4.2 that it is expected that there will be $M - 1 = 3$ small eigenvalues as observed computationally. We consider the dependence of the smallest eigenvalue on ϵ and δt in Table 3.4. Note that these results give evidence that the smallest eigenvalue scales like $\epsilon/\delta t$,

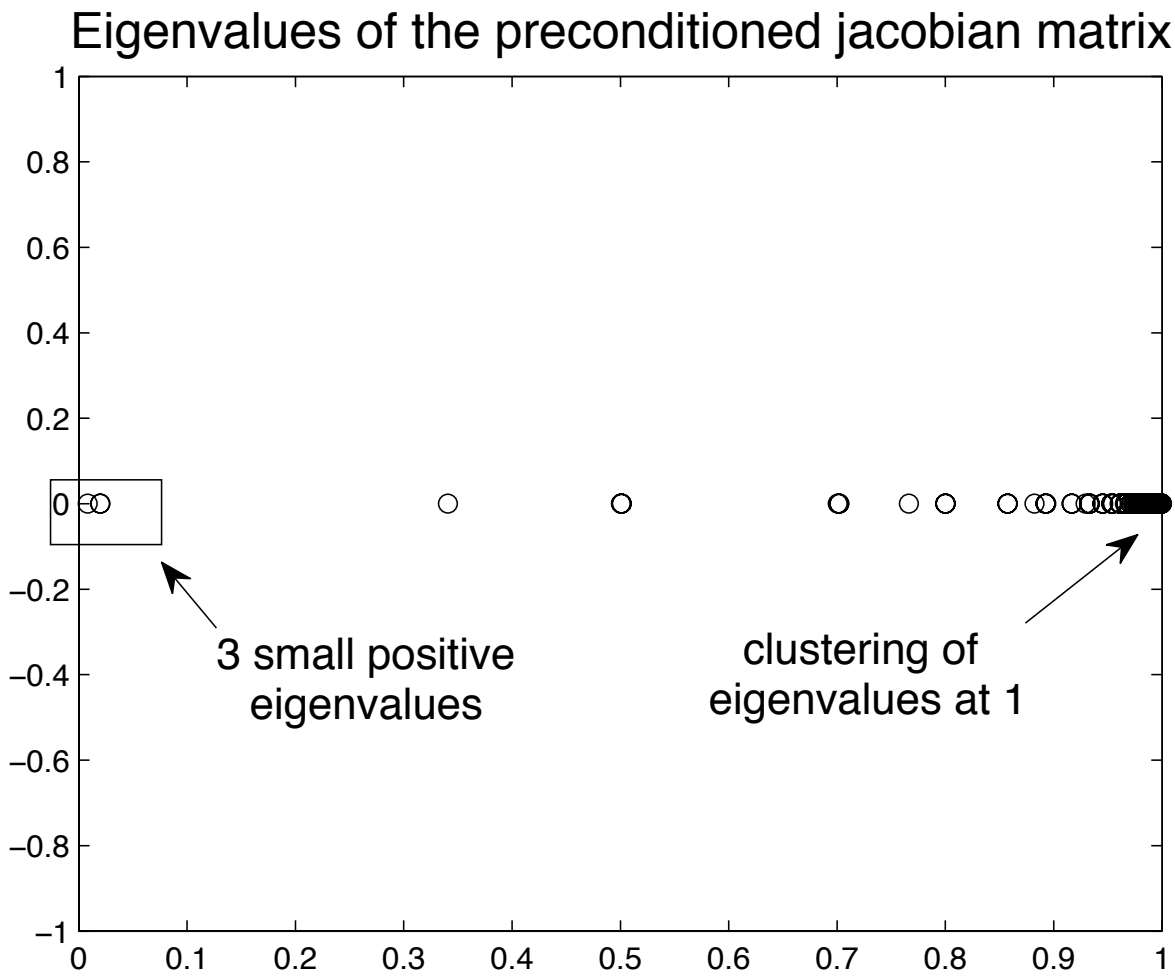


Figure 3.6 Eigenvalues of the preconditioned Jacobian matrix for the model situation described in Section 3.4 for parameters $\epsilon = 0.18$ and $\delta t = 10$.

giving a condition number κ of A that scales like $\delta t/\epsilon$. This is confirmed in the formal asymptotics below.

A similar study with the preconditioned Jacobian for the Allen-Cahn equations reveals M small eigenvalues of identical magnitude that are ndependent of ϵ and have values approximately $C/\delta t$ with $C \approx 0.41$ for large δt. This behavior is confirmed in the formal asymptotics below.

Remark 3. *Since the condition number κ scales linearly with δt and the number of CG*

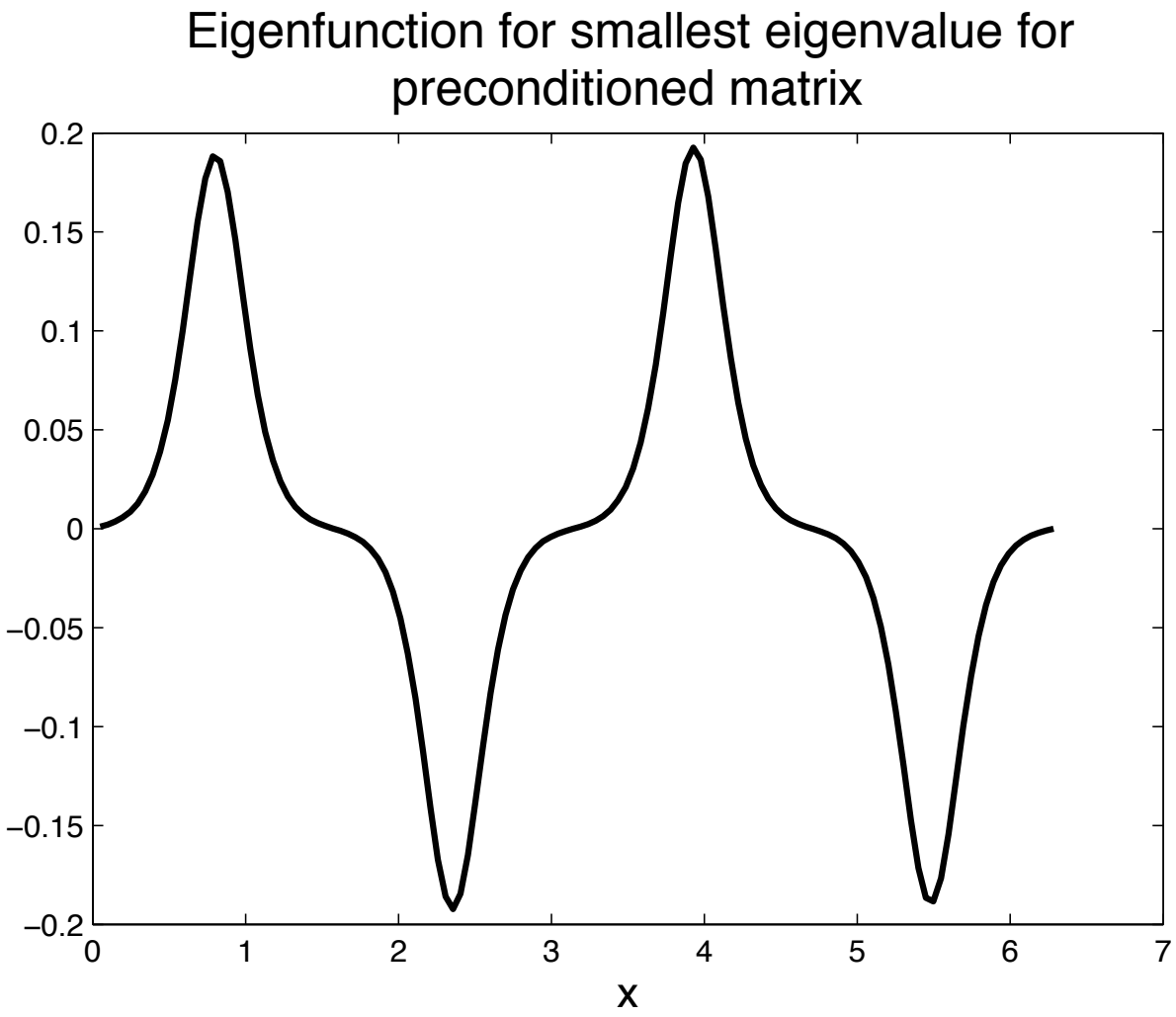


Figure 3.7 The eigenfunction of the preconditioned Jacobian matrix with the smallest eigenvalue for the model situation described in Section 3.4 for parameters $\epsilon = 0.18$ and $\delta t = 10$.

$\delta t\downarrow$ T $\rightarrow$	$\epsilon = 0.18$ 300	$\epsilon = 0.16$ 2000	$\epsilon = 0.14$ 10000
25	3.36e-3	3.14e-3	2.83e-3
50	1.68e-3	1.58e-3	1.45e-3
100	8.43e-4	7.89e-4	7.26e-4
200	4.22e-4	3.95e-4	3.63e-4
400	2.11e-4	1.97e-4	1.82e-4

Table 3.4 Dependence of the smallest eigenvalue of the preconditioned Jacobian matrix for the Cahn Hilliard problem on δt and ϵ. The time T at which eigenvalues are evaluated is roughly where the slowest evolution occurs. The values are not that sensitive to the value of T.

iterations to reach a fixed solution accuracy increases as $\sqrt{\kappa}$ [90], taking larger time steps in ripening regimes will lead to higher numerical efficiency. Thus, we advocate taking as large time steps δt as accuracy will allow. This is a further motivation for our consideration of higher accurate time stepping techniques below.

3.4.2 Formal asymptotics

3.4.2.1 Allen Cahn

Consider first the Allen-Cahn equation (3.1). In an infinite spatial domain this problem has a translationally invariant steady solution

$$u_0 = \tanh\left(\frac{x}{\epsilon\sqrt{2}}\right).$$

In the ripening phase, solutions take the form [94]

$$u \approx \sum_{j=1}^{M} u_0[(-1)^j(x - x_j)] \tag{3.16}$$

where $x_j(t)$ are transition layer positions. This is the structure seen in Figure 3.1 with $M = 4$ transition layers (M must be even in our periodic setting). The approximation above is valid up to exponentially small terms in ϵ. We denote such an approximation by $=_e$ in what follows. Linearizing (3.1) leads to

$$v_t = \mathcal{L}v := \epsilon v_{xx} - (3u^2 - 1)v \tag{3.17}$$

where v is the linear disturbance to $u(x,t)$. The spectrum of $\mathcal{L}$ at ripening states (3.16) is well understood:

Theorem 3.4.1 (Carr and Pego 1989 [94])**.** *$\mathcal{L}$ has M exponentially small (in ϵ) eigenvalues λ_j. The rest are negative and bounded away from zero. The eigenfunctions of the small eigenvalues $\phi_j(x)$ are given by*

$$\phi_j =_e \frac{d}{dx}u_0(x-x_j) = \frac{1}{\epsilon\sqrt{2}}\text{sech}^2\left(\frac{x-x_j}{\epsilon\sqrt{2}}\right) \tag{3.18}$$

which in words are spikes of width ϵ centred at the interface location.

The eigenvalues λ_j govern the exponentially slow motion of the fronts $x_j(t)$. The theorem stated in [94] allows for more general potentials $W(u)$ and the results below can be extended to these cases. Consider now the spectrum of the preconditioned Jacobian for the Allen-Cahn problem at ripening states:

$$A\psi = \sigma\psi$$

where $A = Q^{-1}J$ where Q and J are given by (3.9) and (3.10) respectively. This can be rewritten as

$$(I - \delta t\mathcal{L})\psi = \sigma[I - \delta t(\mathcal{L} - 3(u^2-1))]\psi \tag{3.19}$$

where I is the identity operator. Recall that we are interested in the small eigenvalues σ that determine the condition number of the PCG iterations for this problem and that there is computational evidence that these σ are $\text{O}(1/\delta t)$. Thus we make the ansatz $\sigma = \beta/\delta t$ and

consider (3.19) formally in powers of δt:

$$O(\delta t): \qquad \mathcal{L}\psi = 0 \tag{3.20}$$

$$O(1): \qquad \psi = \beta(\mathcal{L} - 3(u^2 - 1))\psi. \tag{3.21}$$

The first equation above forces ψ to be in

$$\mathcal{V} = \text{span}\{\phi_j\}$$

at highest order where the ϕ_j are the eigenfunctions corresponding to small eigenvalues of Theorem 3.4.1. With $\psi \in \mathcal{V}$, (3.25) is satisfied to exponentially small terms in ϵ. Now (3.21) becomes

$$\psi = 3\beta(1 - u^2)\psi$$

We take $\psi = \phi_j \in \mathcal{V}$ and take the L_2 inner product of the equation above with ϕ_j leading to

$$\beta = \frac{1}{3}\frac{\int \phi_j^2}{\int \phi_j^2(1 - u^2)} \tag{3.22}$$

Considering the ripening form (3.16) and the local nature of ϕ_j (3.18), the value of β only depends on the local layer structure up to exponentially small terms and we thus obtain M copies of

$$\beta =_e \frac{1}{3}\frac{\int \text{sech}^4 x}{\int \text{sech}^4(1 - \tanh^2 x)} \approx 0.4167 \tag{3.23}$$

which agrees closely with the computational results in the previous section. Note that so far (3.21) is satisfied only in the projection on $\mathcal{V}$. Correction terms of order $O(1/\delta t)$ in ψ in $\mathcal{V}^\perp$ can be derived that give a complete picture of the formal analysis.

3.4.2.2 Cahn-Hilliard

The analysis of the small eigenvalues for the preconditioned Jacobian matrix for the Cahn-Hilliard problem can be done similarly, with a few additional technical difficulties. Our results here depend on a conjecture on the rank of a modified square distance matrix. Here the relevant eigenvalue problem is

$$(I + \delta t D\mathcal{L})\psi = \sigma[I + \delta t D(\mathcal{L} - 3(u^2 - 1))]\psi \tag{3.24}$$

where D is the second derivative operator. We use D here rather than Δ to be clear that the analysis that follows is only applicable to the 1D case. The extra difficulty here arises essentially because D is not invariant on $\mathcal{V}$ (the span of the eigenfunctions ϕ_j) or $\mathcal{V}^\perp$. We make the ansatz $\sigma = \beta/\delta t$ as before and consider (3.24) formally in powers of δt:

$$O(\delta t): \qquad D\mathcal{L}\psi = 0 \tag{3.25}$$

$$O(1): \qquad \psi = \beta D(\mathcal{L} - 3(u^2 - 1))\psi. \tag{3.26}$$

As before, (3.25) forces $\psi \in \mathcal{V}$ to highest order which we make explicit here

$$\psi = \sum_{j=1}^{M} c_j \phi_j. \tag{3.27}$$

More carefully said, $\mathcal{L}\psi$ must be a constant to satisfy (3.25) but this constant enters as a $O(1/\delta t)$ term. Turning to (3.26) we first notice that ψ is a second derivative of a periodic

function and so must have average zero which gives the discrete condition

$$\sum_{j=1}^{M} c_j =_e 0. \tag{3.28}$$

Let P be the projection onto the set of functions with average value zero:

$$P\phi = \phi - \frac{1}{2\pi}\int_0^{2\pi} \phi(x)dx.$$

Let $\phi_j^* = P\phi_j$. Because of (3.28), (3.27) can be written equivalently as

$$\psi = \sum_{j=1}^{M} c_j \phi_j^*.$$

We can apply D^{-1} (taking the result to have zero average value) to (3.26), obtaining

$$\sum_{j=1}^{M} c_j D^{-1}\phi_j^* = 3\beta P(u^2-1)\sum_{j=1}^{M} c_j \phi_j.$$

Taking the inner product with an arbitrary element of $\mathcal{V}$ with zero mass, then normalizing with $\int \phi_j^2 = K/\epsilon$ (consider the form (3.18) to see that this gives an absolute constant K) we obtain

$$-\beta_{AC}\frac{\epsilon}{K} PBP\underline{c} = \beta \underline{c} \tag{3.29}$$

where $\underline{c}$ is the vector of M values c_j, P is here the discrete projection onto the subspace of vectors that sum to zero, β_{AC} is the constant from the Allen-Cahn eigenvalue problem

(3.23), and B is the $M \times M$ symmetric matrix with entries

$$b_{ij} = \int \phi_i^* D^{-1} \phi_j^*. \tag{3.30}$$

As above for the simpler Allen-Cahn case, (3.26) is so far only satisfied in the massless subspace of $\mathcal{V}$. Full matching can be done with $O(1/\delta t)$ terms in $\mathcal{V}^\perp$ plus constants. Considering (3.29), the expected behavior of the small preconditioned Jacobian eigenvalues of size $\epsilon/\delta t$ will be observed as long as PBP is full rank $M-1$ and has $O(1)$ eigenvalues. Consider (3.30) in the limit as $\epsilon \to 0$. The integral $\int \phi_j =_e 2$ remains fixed in this limit and so ϕ_j is seen to be an approximate delta function at x_j with weight 2. Thus, $D^{-1}\phi_j^*$ is approximately a quadratic with second derivative $-1/\pi$ except in a region of width ϵ around x_j:

$$D^{-1}\phi_j^*(x) \approx d - \frac{d^2}{2\pi} - 2\pi + \frac{4\pi^2}{3}$$

where d is the distance between x and x_j on the periodic interval $[0, 2\pi]$. The integral (3.30) can then be approximated by

$$b_{ij} \approx 2d_{ij} - d_{ij}^2/\pi \tag{3.31}$$

where d_{ij} is the distance between points x_i and x_j on the periodic interval and we have adjusted the entries of B by a constant as allowed by the expression (3.29) to simplify the expression. Note that the expression above is only accurate to polynomial terms in ϵ. We have strong computational evidence that the matrix PBP with limiting entries (3.31) has rank $M-1$ whenever the layer positions x_i are distinct. Thus, we conjecture

Conjecture 1. *If $\{x_i\}, i = 1, \ldots M$ are distinct points in $[0,1]$ and the entries b_{ij} of the*

$M \times M$ matrix B are given by

$$b_{ij} = d_{ij} - d_{ij}^2$$

where d_{ij} are the distances between points x_i and x_j, taken either as absolute values or on the periodic interval, then the matrix PBP has $M-1$ strictly negative eigenvalues. Here P is projection onto the subspace orthogonal to constant vectors.

In the statement above we have scaled the interval to $[0, 1]$ for convenience. It should be noted that without the first term in b_{ij}, linear in distance with scaling proportional to the interval size, the square distance matrix PBP has rank at most 3 [96].

3.5 Performance on a variety of models

One of the main advantages of our approach is that it is easily extensible to higher order and vector models. The Jacobian matrix and preconditioner are straight forward to generate for these more complicated models. There is no need to determine how to split the potential into convex and concave pieces, as is needed for the popular class of splitting methods based on Eyre's method [46] which we discuss in section 3.9 below. While we do not claim that our approach is the most computationally efficient methods for every (or any) problem, they are reasonably efficient and can be tried on any new problem with little effort. Here, we apply them to the Cahn-Hilliard problem (3.2), the sixth order model problem (1.10) and the vector model problem (3.5), all in 2D. We conduct these numerical tests with first order time stepping for simplicity.

In the examples below, convergence in ripening times and the behavior of the number of time steps and total PCG iterations with tolerance σ is observed as in the preliminary 1D Cahn-Hilliard example shown in section 3.3.4 .

3.5.1 2D Cahn-Hillard

We begin with initial conditions

$$u_0(x, y) = 2e^{\sin x + \sin y - 2} + 2.2e^{-\sin x - \sin y - 2} - 1 \tag{3.32}$$

which after some initial ripening of $u = 1$ states (mass fraction 0.4556) leads to two circular states, one of which captures the other in a long time frame. The results for the $\epsilon = 0.08$ model, using $N = 128$ and $\sigma = 10^{-4}$ are shown in Figure 3.8. This simulation took 3,305 time steps and a total of 84,138 PCG iterations. The similar computation for $\epsilon = 0.16$ using $N = 64$ and $\sigma = 10^{-4}$ past ripening took 1,471 time steps and a total of 30,051 PCG iterations. The MATLAB code used for this example will be available on the publisher's web site for this article.

3.5.2 Sixth order model

We begin with the same initial conditions (3.32) shown in Figure 3.8 upper left. The results for $\epsilon = 0.18$, $\eta = 1$ computed with $N = 128$ and $\sigma = 10^{-4}$ are shown in Figure 3.9. The influence of the interface promoting term is evident. For larger ϵ the final steady state is a regular array. The final pattern shown here is not at steady state. This simulation was done in 2,229 time steps with a total of 234,582 PCG iterations.

3.5.3 2D vector model

We begin with initial conditions (3.32) for u and $v_0(x, y) = \sin y$. After some initial ripening, regions of the three states form, separated by interfaces which meet at triple junctions. Unlike grain growth [93] this model preserves the area of each of the three phases. The results for

the $\epsilon = 0.32$ (the ϵ value compared to other models is larger due to the larger magnitude of the reaction term for this model), using $N = 128$ and $\sigma = 10^{-4}$ are shown in Figure 3.10. This simulation took 2,229 time steps and 51,985 total CG iterations. We believe that the final result at $t = 100$ is an approximation of the steady state of the problem.

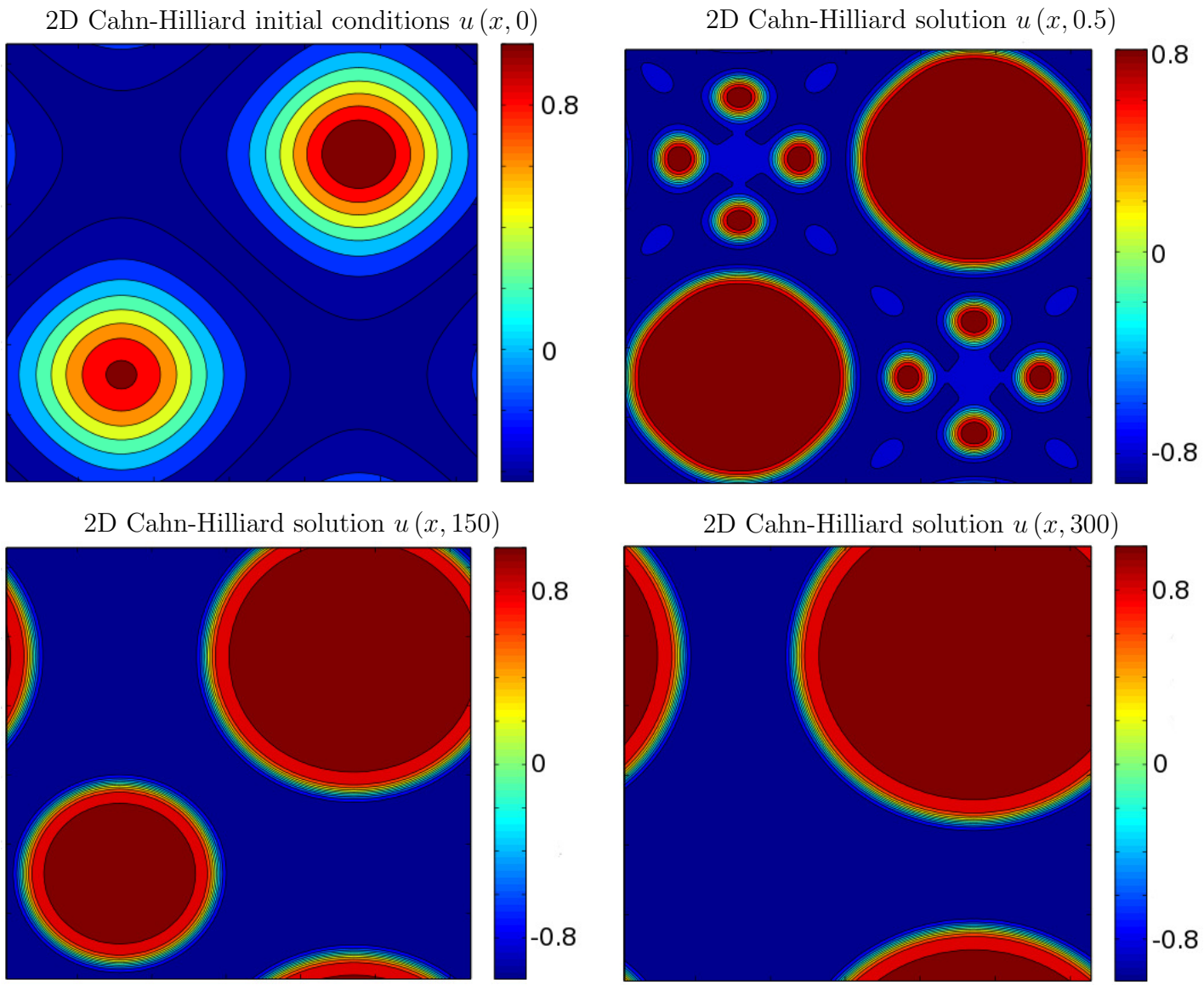


Figure 3.8 2D Cahn-Hilliard example computation.

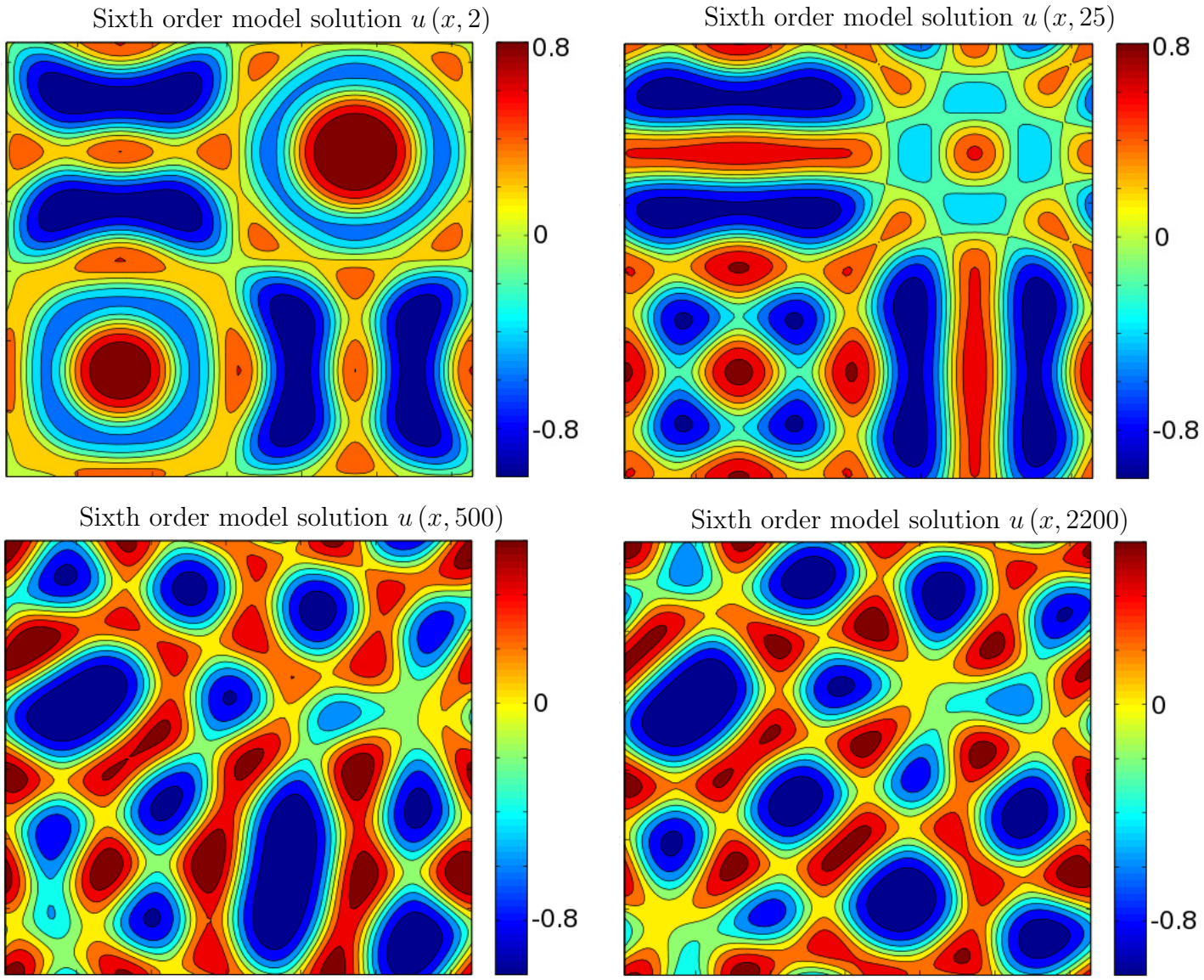


Figure 3.9 2D sixth order model example computation.

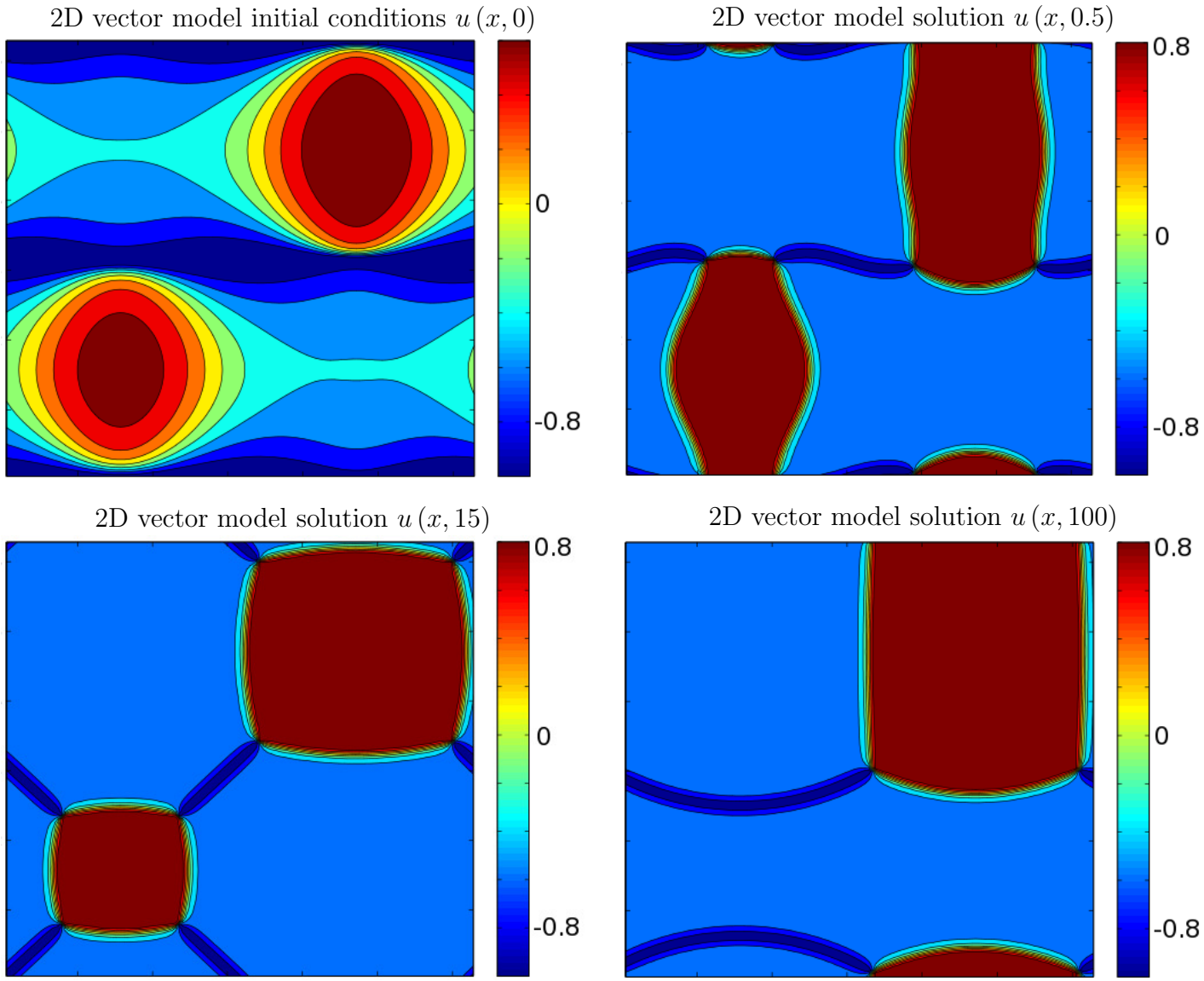


Figure 3.10 2D vector model example computation. Contours of $\cos(\arg u + iv)$ are plotted. Two of the phases have value $\cos(2\pi/3) = -1/2$ (light blue in the plots) and are separated by dark blue lines.

3.6 GPU implementation

A recent development in scientific computing is General-Purpose computing on Graphics Processing Units (GPGPU). In the 1990s, Graphics Processing Units (GPUs) were developed to accelerate the building of 2D and 3D images for output to a display. Since then, the stream processing capabilities of GPUs have been turned to scientific computing, and the application to scientific computing has led to commercial lines of GPUs that have hundreds of double precision computation cores with error checking. GPU computation can be much faster if a problem or computation lends itself well to many lightweight computation threads. We used Nvidia GPU cards and the CUDA programming language, which is C++ with extra functions to control the device (GPU) from the host (CPU).

When simulating the Cahn-Hilliard equation implicitly with the spectral method detailed in Section 3.3, there are three main types of mathematical operations: Fourier transforms, element-wise multiplication or addition, and array reductions (the inner products in the PCG method for example). The first two of these operations adapt very well to GPU processing, but array reductions pose some challenges.

Three-dimensional Fourier transforms can be computed extremely fast on a GPU by breaking the domain into lines of grid points, then handing each line of grid points to a separate multi-processor for calculation of the one dimensional FFT. After each processor is done computing the frequency representation of the data, the GPU recompiles the data into its three-dimensional representation. The widely available cuFFT package implements FFT calculations on the GPU. With the solution available in both spatial and frequency domains, multiplication by the Jacobian matrix and the preconditioner can be performed by element-wise multiplication or addition in the appropriate domain. This plays directly into

the strength of the GPU which has millions of lightweight computation threads.

The third type of operation performed when computing solutions to the Cahn-Hilliard equation is array reductions. Reductions must be used when computing the inner products as well as maximum residual values for the conjugate gradient and Newton method iterations. In a CPU calculation, array reductions are performed serially so they are simple to implement and run quickly compared to other parts of the calculation. However, array reductions are particularly troublesome for GPUs. Besides having hundreds of processors, GPUs are fast because they hide latency by scheduling calculations in a first come first served manner. Thus it is impossible to know before computation time in which order the 2^{21} evaluations will take place. The solution to this problem is to synchronize all threads after each set of evaluations, even though setting such a thread block increases computation time. An example of the proper way to approach array reduction on GPUs is given in the CUDA SDK provided by Nvidia.

There are limitations to GPU computing that cannot be overlooked. Unlike a CPU that has access to any memory location, GPUs multiprocessors can only access the data that is stored in the GPU card's memory. This leads to two complications: limitations on array size and movement of data. The GPU card we used had 6GB of memory on the card, so for the number of double-precision, complex-valued arrays we used in the simulation, the size of the domain was limited to 2^{22} total grid points. For a cubic domain, this limit corresponds to 128 grid points in each dimension. It is important to realize that this is a rigid memory limit because it is defined by the GPU card's hardware. It is certainly possible to increase the domain size by using multiple GPUs for a computation, but this would require significant effort to reduce the data movement between domains that would limit the computational speedup. The second complication is movement of the data into the memory on the GPU.

The movement of data on and off the card is the only way to access the solutions to the PDE, and data movement is much slower than computation speed. Because of this, the number of times the solution array is moved from the card must be kept at a minimum. This places some limitations on the post-processing of solutions.

3.6.1 GPU Speedup

We conducted a speed test to compare simulation code written for parallel CPUs against the code written for the GPU. The CPU code was written in C++ and aggressively parallelized using OpenMP. The parallel version of FFTW was used for the Fourier transforms. The code was run on two Quad-core Intel Xeon E5620 processors (eight total cores). The processors have a clock speed of 2.4 GHz. The GPU used was a Tesla C2070 with 448 multiprocessor cores with a clock speed of 1.15 GHz, and it had 6GB of global memory. The two hardware configurations were about one year old at the time of the test, so the speed results are comparable. The test was performed on initial conditions of 3D random initial data on cubic grids with 128 grid points per dimension. Computing to time $T = 40$, the parallelized C++ calculation took nine hours and seventeen minutes to complete compared to the Cuda GPU code that took one hour and twenty-five minutes. The solutions given by the two codes were identical. We realized a speedup factor of about 6.5 with the GPU implementation over the eight core CPU calculation in this computation in which data transfer on and off the GPU card was negligible. This speedup factor was confirmed in timing tests of individual time steps. The computed solutions are shown in Figure 3.11 and the time step and PCG profiles are shown in Figure 3.12 and Figure 3.13, respectively. The sequential decreases in time step size correspond to ripening events.

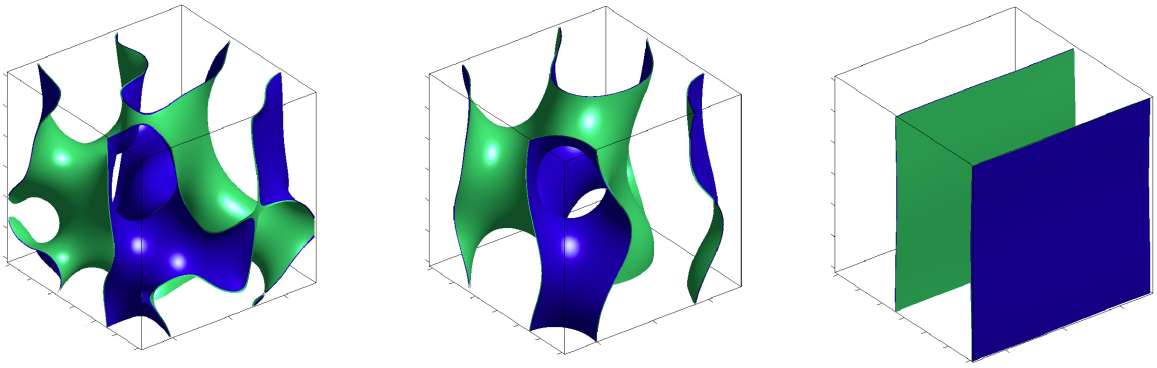


Figure 3.11 Solution at times $T = 5$, 10, and 40 of the 3D Cahn-Hilliard computation used as a timing test for the GPU implementation. Shown are the zero level sets of the solutions, with blue indicating increasing and green decreasing solutions at the interface.

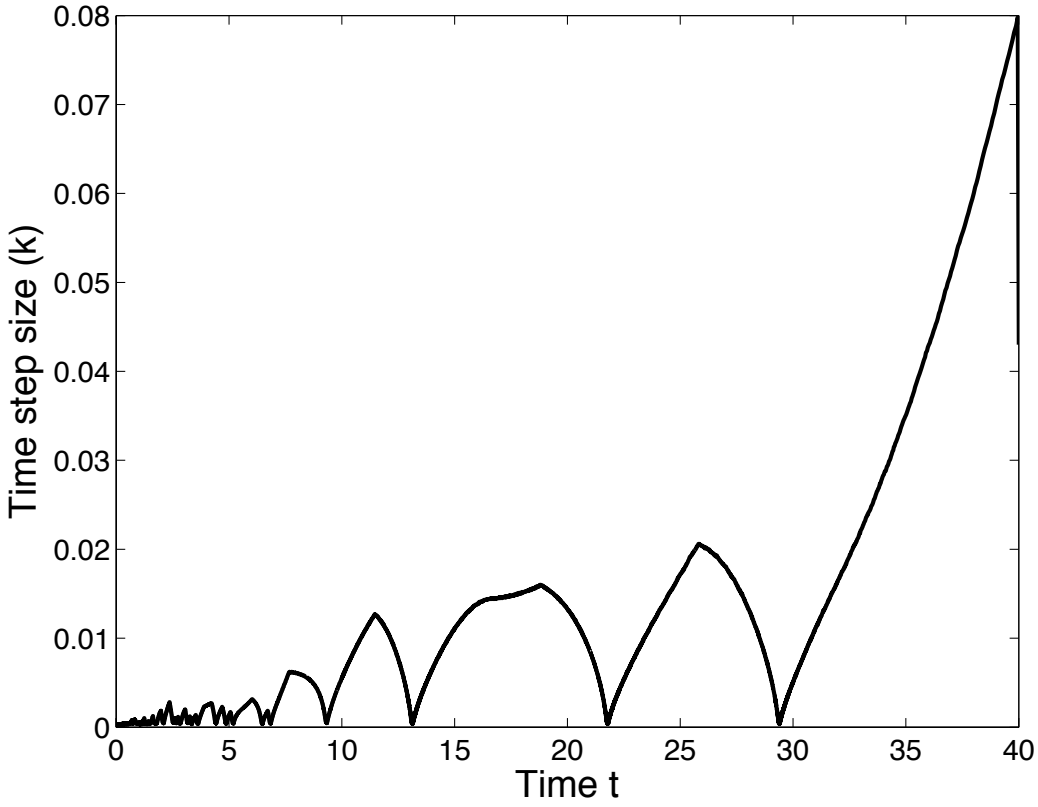


Figure 3.12 Time step size δt for the 3D Cahn-Hilliard computation used as a timing test for the GPU implementation.

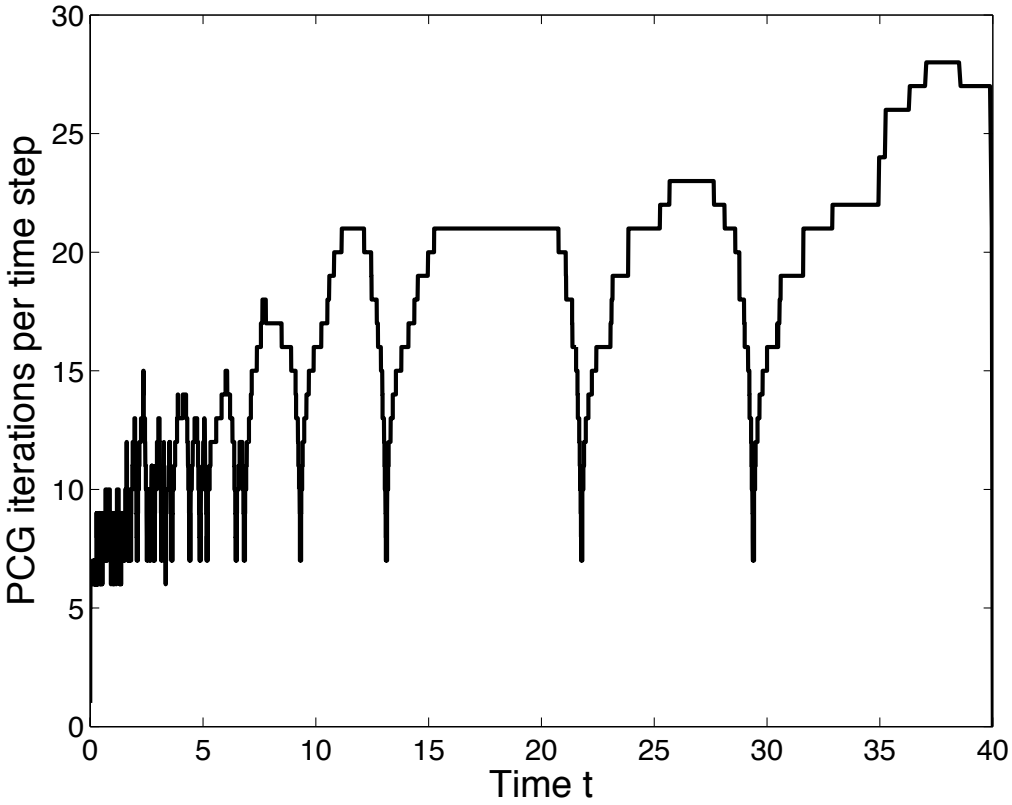


Figure 3.13 PCG count per time step for the 3D Cahn-Hilliard computation used as a timing test for the GPU implementation.

3.7 Higher order time stepping

We now consider second and third order adaptive time stepping methods. We use the multistep methods Adams-Bashforth (AB) and Backward Difference Formula (BDF) as our explicit predictor and implicit corrector respectively. With the same notation for the fully discrete approximation of u given in section 3.3.2, the second and third order versions of AB for the predicted $\mathbf{U}^*$ solution at time t_m are [89]:

$$\begin{aligned}
\text{(AB2): } \mathbf{U}^* &= \mathbf{U}^{m-1} + \frac{\delta t_m}{4}\left[6f(\mathbf{U}^{m-1}) - f(\mathbf{U}^{m-2})\right], \\
\text{(AB3): } \mathbf{U}^* &= \mathbf{U}^{m-1} + \frac{\delta t_m}{12}\left[23f(\mathbf{U}^{m-1}) - 16f(\mathbf{U}^{m-2}) + 5f(\mathbf{U}^{m-3})\right],
\end{aligned}$$

where for example for the lower order Allen-Cahn equation,

$$f(u) = \epsilon^2 \Delta_h u - W'(u).$$

Again we use these as an initial state in a Newton iteration for the solution to the implicit BDF time step. The second and third order versions of BDF lead to the nonlinear systems $G(\mathbf{U}^m)$ for the solution at time t_m given below [88]:

$$\begin{aligned}
\text{(BDF2): } G(U^m) &:= \frac{3}{2}U^m - \delta t_m f(U^m) - 2U^{m-1} + \frac{1}{2}U^{m-2} = 0, \\
\text{(BDF3): } G(U^m) &:= \frac{11}{16}U^m - \delta t_m f(U^m) - 3U^{m-1} + \frac{3}{2}U^{m-2} - \frac{1}{3}U^{m-3} = 0.
\end{aligned}$$

The solution procedure for these systems is the same as described in section 3.3.3. The combination of AB predictor and BDF corrector has several advantages. BDF methods up to order 6 have good stability properties for stiff problems and have a high ratio of accuracy

to implicit solves [88]. Like forward and backward Euler methods, AB and BDF methods of the same order p share the same form of dominant local truncation error, proportional to the order $p+1$ time derivative of u. Because of this, the exact local truncation error η_e for the second and third order steps can be easily approximated by

$$\eta_e \approx \eta := C||U^* - U^m|| \tag{3.33}$$

where C is $4/9$ and $2/5$ for order 2 and 3 methods, respectively. These higher order methods require p previous values for order p methods. Standard AB and BDF methods require these previous values to be equally spaced in time. This is the major drawback of standard multi-step methods. Initially and after every time step change, we compute the necessary additional previous values using minimal stage L-stable Singly Diagonal Implicit Runge Kutta (SDIRK) methods of second or third order [97].

In our time-adaptive computations below, we again specify a given tolerance $\sigma > 0$ for the local truncation error η. Because changing the time step requires additional computational computational cost we only increase the time step if the local error is sufficiently below tolerance,

$$\frac{\eta}{\sigma} < \frac{1}{\gamma}$$

where $\gamma > 1$ is user defined. We chose $\gamma = 3$ based on computational evidence [98]. Total time step counts are relatively insensitive to γ around this value. If the condition above is met, we allow a time step increase with multiplier

$$\xi = (0.8\gamma)^{1/(p+1)}.$$

where again 0.8 is a safety factor. If a time step fails the accuracy check ($\eta < \sigma$) then we fail the step and reduce the time step by a factor of $\frac{1}{\xi}$ and restart the computation.

Although SDIRK steps are more accurate than BDF steps for the same time step size, the errors are different and so the error estimator (3.33) cannot be used directly after a restart. However, the error mismatch decays with a fixed number of steps in a numerical initial layer effect. Because of this, we only check the error after a preset number S of time steps following a restart for both an increase and decrease in the time step. The values of S used are 4 and 8 for BDF2-AB2 and BDF3-AB3 respectively (the numerical initial layers for BDF3 have a slower decay than for BDF2).

3.7.1 Numerical Results

We consider the same 1D Cahn-Hilliard model problem as in section 3.3.4, where the initial data $u_0(x)$ is given by (21) and $\epsilon = 0.18$. In these runs the Newton solve has residual tolerance 10^{-9} and the PCG solve at each step has tolerance 10^{-10}. The time step size and PCG iteration history are shown in Figure 3.14. Note that the time step history for the first order method (FE/BE) is different in Figure 3.14 than in Figure 3.5 because we apply the same accuracy threshold to changing time steps here that we do for the higher order methods.

3.8 A Pair of Fourth-Order Accurate Methods

In recent work with Dr. David Seal, we discovered and implemented a new family of high-order time-stepping Lax-Wendroff schemes. The fourth-order implicit method is A-stable and effectively avoids the Dahlquist barrier by using the derivatives of the right hand side of

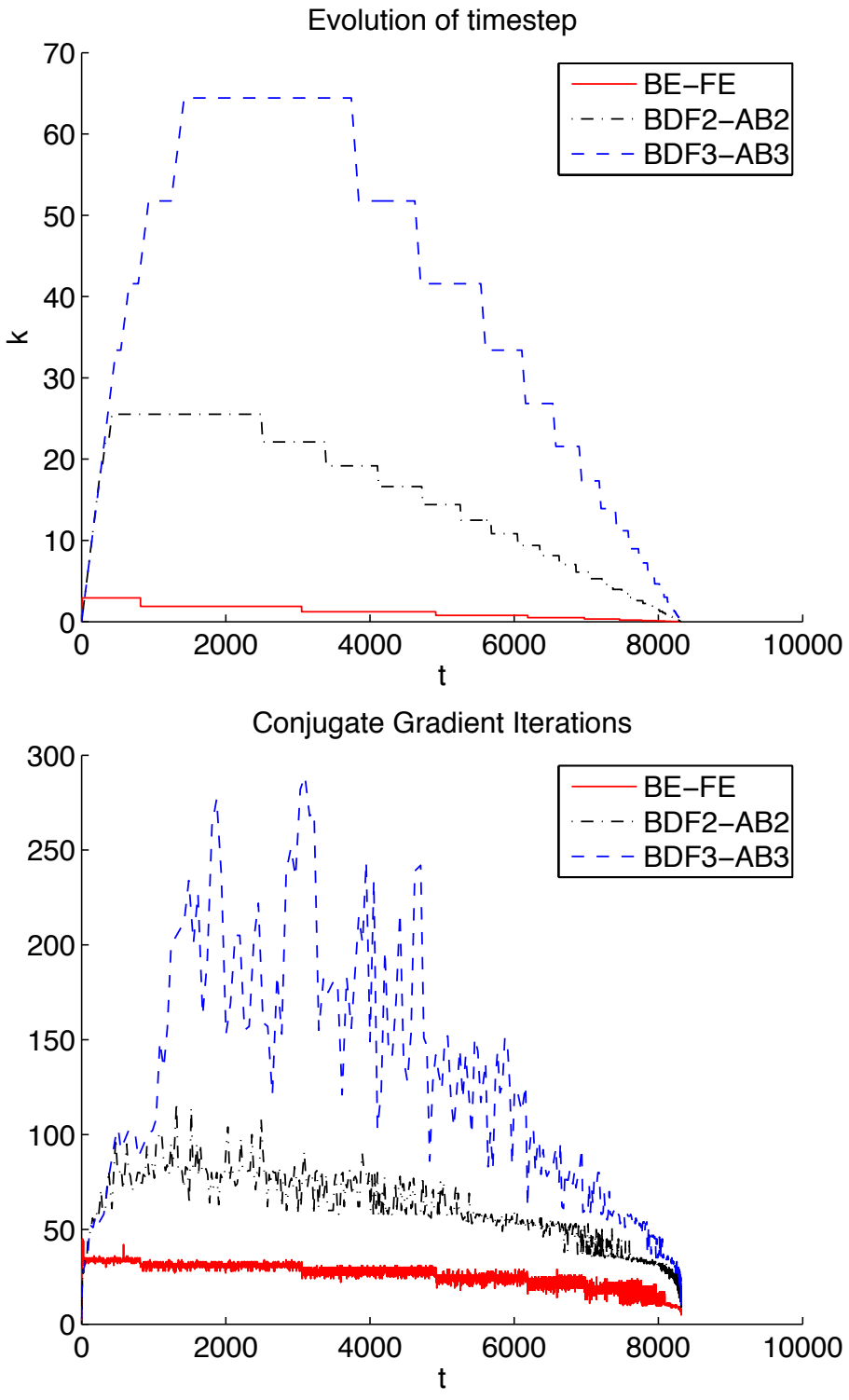


Figure 3.14 The time step and PCG iteration count history for the adaptive time approximation of the 1D Cahn-Hilliard using BE-FE, BDF2-AB2 and BDF3-AB3 with initial conditions (21) for $\epsilon = 0.18, \sigma = 10^{-7}$.

the equation. The fourth-order explicit method has the same error term as the corresponding implicit scheme allowing an adaptive time-stepping procedure.

For the general PDE,

$$u_t = f(u)$$

the two-stage explicit scheme is given by

$$\begin{aligned} u^* &= u^n + \frac{\delta t}{2} f(u^n) + \frac{\delta t^2}{8} \mathcal{J} f(u^n) \\ u^{n+1} &= u^n + \delta t f(u^n) + \frac{\delta t^2}{6} \left(\mathcal{J} f(u^n) + 2 \mathcal{J} f(u^*) \right), \end{aligned} \tag{3.34}$$

and the implicit scheme is

$$u^{n+1} = u^n + \frac{\delta t}{2} \left(f(u^n) + f\left(u^{n+1}\right) \right) + \frac{\delta t^2}{12} \left(\mathcal{J} f(u^n) - \mathcal{J} f\left(u^{n+1}\right) \right), \tag{3.35}$$

where $\mathcal{J}$ is the Jacobian of the right-hand side of the differential equation, f.

For the CH equation, $f = -\varepsilon^2 \Delta^2 u + \Delta\left(u^3 - u\right)$, so the Jacobian acting on some object v is $\mathcal{J} v = -\varepsilon^2 \Delta^2 v + \Delta\left[\left(3u^2 - 1\right) v\right]$. We compute the explicit solution and then solve for the implicit solution as above. The adaptive time-step is computed in the same manner also, except

$$\delta t_{new} = \delta t_{old} \cdot \min\left(0.8 \cdot \sqrt[5]{\frac{\sigma}{E}}, 1.3 \right). \tag{3.36}$$

The preconditioner we use is derived in the same way as the first order preconditioner. For the fourth order method applied to the CH equation,

$$G'(u^r)\, v = v - \frac{\delta t}{2} \mathcal{J} f(v) - \frac{\delta t^2}{12} \mathcal{H} f(v), \tag{3.37}$$

with $\mathcal{H}f = \frac{\partial u_{ttt}}{\partial u}$ computed by hand. Assuming $u = \pm 1$ and $\Delta u = 0$ for the majority of the domain, we obtain the physics based preconditioner,

$$Q = 1 - \frac{\delta t}{2}\left(-\varepsilon^2\Delta^2 + 2\Delta\right) + \frac{\delta t^2}{12}\left(-\varepsilon^2\Delta^2 + 2\right)\left(-\varepsilon^2\Delta^2 + 2\Delta\right). \tag{3.38}$$

For the Cahn-Hilliard equation, we observe fourth order convergence for this pair of methods. The physics based preconditioner made a significant difference by reducing the required number of CG steps by a factor of one hundred throughout most of the computation. During the spinodal phase separation exhibited by solutions of the CH equation (computed from random initial data), time steps for the fourth order method were ten to a thousand times larger than the first order implicit method (Section 3.3.2). When the solution became smooth (or if smooth initial conditions were used), the first and fourth order methods maintained approximately the same size time steps when controlling the error. This comes from the fact that when using a fixed time step, the first order method was numerically stable for a time step size slightly larger than the fourth order method. The Newton and conjugate gradient solves required nearly the same number of iterations per time step, but as expected, the fourth order method required more computation per iteration.

The results in Table 3.5 shows the performance of the methods on the test problem for various values of local tolerance σ. Ripening times can be computed very accurately with the higher order methods. The PCG count is an accurate measure of computational cost and includes all iterations from SDIRK restart and failed steps. Efficiency gains are obtained with the higher order methods, with the biggest gain moving from first to second order. As in section 3.3.4 the first order method shows the correct asymptotic behavior in the number of time steps (ratios of $\sqrt{10} \approx 3.16$ of times steps as σ is decreased by 10). The second order

method results are also consistent with the correct asymptotic behavior ratio $\sqrt[3]{10} \approx 2.15$. The third order results have not reached the asymptotic regime. Considering the accuracy of the ripening times in this case, it appears that the error estimation is *over-estimating* the local error.

Remark 4. *Experiments with other higher order strategies were also done, for example using SDIRK2 stepping with SDIRK3 for error estimation. However, for tolerances σ leading to accuracy of practical interest, this error estimation was also not asymptotically valid. It is known that constructing good error estimators for IRK methods applied to very stiff problems is difficult [99]. We conjecture that the relatively large higher derivative terms pollute the error estimation in this case. The problem of accurate error estimation for higher order time stepping for this class of problems is an interesting open question. With such error estimation, arbitrary precision could be achieved to solutions of these problems: spectral accuracy in space and high accuracy in time using spectral deferred correction methods [100] or high order Radau methods [88].*

3.9 Investigation of splitting methods

3.9.1 Preliminaries and numerical results

We present a splitting approach of Eyre [46] in the framework of the implicit approximation of the 1D Cahn-Hillard model. The standard backward Euler time stepping approximation of this model leads to the problem

$$\mathbf{U}^m + \delta t \Delta_h \left[\epsilon^2 \Delta_h \mathbf{U}^m - (\mathbf{U}^m)^{<3>} + \mathbf{U}^m \right] - \mathbf{U}^{m-1} = \mathbf{0} \tag{3.39}$$

Method	σ	time steps	ripening time	total PCG iterations
BE-FE	1e-04	792	8206.11	41475
	1e-05	2114 (2.67)	8272.16	69166 (1.67)
	1e-06	6371 (3.01)	8303.81	125879 (1.82)
	1e-07	20216 (3.17)	8314.15	277079 (2.20)
BDF2-AB2	1e-04	191	8301.34	22663
	1e-05	660 (3.455)	8316.19	36535 (1.612)
	1e-06	1484 (2.248)	8317.97	44987 (1.231)
	1e-07	3279 (2.210)	8318.47	70580 (1.569)
BDF3-AB3	1e-06	374	8318.43	25443
	1e-07	981 (2.623)	8318.61	37582 (1.477)
	1e-08	2041 (2.081)	8318.63	53079 (1.412)

Table 3.5 Performance of each adaptive time stepping method through a ripening event of the 1D Cahn-Hilliard model with $\epsilon = 0.18$ and $N = 128$. The numbers in brackets are ratios to quantities in the previous row.

where by $(\mathbf{U}^m)^{<3>}$ we mean the pointwise values cubed and δt is the time step as before. This is the system solved with the methods in this chapter. A slight modification to this system, taking explicitly the linear term that otherwise makes the system non-convex, Eyre [46] considered

$$\mathbf{U}^m + \delta t \Delta_h \left[\epsilon^2 \Delta_h \mathbf{U}^m - (\mathbf{U}^m)^{<3>} \right] - \mathbf{U}^{m-1} + \delta t \Delta_h \mathbf{U}^{m-1} = \mathbf{0}. \tag{3.40}$$

Note that the implicit solve is still nonlinear but now convex. There are two attractive theoretical properties to this approach:

A: The discrete problem (3.40) has a unique solution for every δt and every U^{m-1}.

B: The unique solution above is guaranteed to reduce the energy.

The nonlinear system can be solved with the same Newton, PCG approach as the unsplit discretization we have considered up to this point and we can examine the condition number of the preconditioned Jacobian matrix in the same framework as the preconditioned unsplit

method in section 3.4. Here, the matrix of interest is

$$A_S = Q_S^{-1} J_S$$

where $Q_S = I + \delta t \epsilon^2 \Delta_h \Delta_h - 3\delta t \Delta_h$ and

$$J_S = I + \delta t \epsilon^2 \Delta_h \Delta_h - \delta t \Delta_h \Lambda_S$$

where Λ_S is the diagonal matrix with positive entries $3[U_j^{(r)}]^2$ where $\mathbf{U}^{(r)}$ denotes the r'th iterate for $\mathbf{U}^m$ as before. Note that J_S is positive definite for every δt and $\mathbf{U}^{(r)}$ (a consequence of the convexity of the implicit problem), unlike J_{CH} (3.11) from the unsplit discretization. The eigenvalues of A_S like those of A in section 3.4 also cluster around 1 and have a minimum value that determines the condition number of A_S. Here, the minimum eigenvalue for large δt at ripening states is approximately 0.290 *independent of* δt *and* ϵ. This matches the value predicted by formal asymptotics shown in section 3.9.2 below.

It is possible to set up a linear fixed point iteration and avoid the machinery of newton iterations with PCG solvers. Consider the iterative scheme with iterates $\mathbf{U}^{(r)}$ for $\mathbf{U}^m$ to solve (3.40):

$$\begin{aligned} \mathbf{U}^{(r)} + \delta t \Delta_h \left[\epsilon^2 \Delta_h \mathbf{U}^{(r)} - 3\mathbf{U}^{(r)} \right] &= \delta t \Delta_h \left[(\mathbf{U}^{(r-1)})^{<3>} - 3\mathbf{U}^{(r-1)} \right] \\ &\quad + \mathbf{U}^{m-1} - \delta t \Delta_h \mathbf{U}^{m-1} \end{aligned}$$

This approach is taken in [91]. Note that the linearization of the fixed point iteration has matrix $I - A_S$, so the small eigenvalues of A_S will determine the convergence rate of the iterations. The convergence factors will be approximately 1-0.290 = 0.710 for the 1D C-H

ripening solutions considered in this section, independent of δt and ϵ. Eyre also considered other operator splitting possibilities, including one that only requires a linear implicit solve

$$\mathbf{U}^m + \delta t \Delta_h \left[\epsilon^2 \Delta_h \mathbf{U}^m - 4\mathbf{U}^m \right] - \mathbf{U}^{m-1} + \delta t \Delta_h (4\mathbf{U}^{m-1} - (\mathbf{U}^{m-1})^3) = \mathbf{0} \tag{3.41}$$

which retains properties **A** and **B** above under the assumption that $\|\mathbf{U}\|_\infty$ remains bounded by 1.

From the point of view of solver efficiency, the splitting methods are ideal. However, the accuracy of time stepping with the operator splitting can be much lower than that of the conventional (non-split) solvers considered in this chapter. We repeat the 1D Cahn-Hilliard ripening time computations of section 3.3.4. The results are shown in Table 3.6 for the split step method (3.40) using the PCG solver for the time stepping. Comparing to Table 3.3, it is clear that the splitting method is *much* less efficient than the pure implicit time stepping proposed in this chapter. It is clear from Table 3.6 that this is not just an artifact of the error estimation, since even with the large number of time steps, ripening times are not found accurately. The linear splitting (3.41) performed similarly poorly, taking 193,673 time steps for $\sigma = 10^{-4}$ in the same computation listed in Tables 3.3 and 3.6, obtaining a (very inaccurate) ripening time of 15301. Although the solve at each time step is more efficient and the method has the same order of accuracy, the errors from the operator splitting are much larger than those from pure implicit time stepping for large time steps. Poor performance of splitting methods is observed in 2D computations and in the other models considered in section 3.5. For example, using Eyre's method (3.40) on the 2D Cahn-Hilliard computation described in section 3.5 leads to 83,862 time steps and 1,069,416 PCG iterations for the $\epsilon = 0.08$ run (compared to 3,305 and 84,138 respectively for our fully implicit approach)

and 18,891 time steps and 207,252 PCG iterations for the $\epsilon = 0.16$ case (compared to 1,471 and 30,051 for our approach). The accuracy limitations of splitting methods has previously been observed [55, 91]. Using higher order implicit-explicit (IMEX) time stepping methods [101, 102, 103, 104] (which retain property **A** above but not necessarily **B**) does not allow significantly larger time steps through ripening, although they achieve the specified order of accuracy as $\delta t \to 0$ of course. The accuracy loss can be explained in the 1D Allen-Cahn case using simple asymptotics below.

Remark 5. *The recent method in [55] is different from Eyre's splitting ideas. Their technique ensures property* **B** *above but not* **A**. *The implicit problem they solve at every time step (with incomplete LU preconditioned GMRES) has the same structure as the fully implicit time stepping considered in the current work, although it is a modification of the implicit midpoint rule which does not have strong stability properties. We believe our PCG solution strategy could be applied to their discretization and should be more efficient.*

Remark 6. *It should be made clear that these 1D problems with exponentially slow ripening events are extreme cases. It is not clear whether Eyre's type splitting methods might be made more efficient in some 2D and 3D problems where motion is generically only polynomial slow in ϵ. If one relaxes the strict requirement of time accuracy and considers a computational time step as just an efficient step down the energy landscape[91], the comparison is even less clear. See also [105, 106] for time stepping strategies that preserve coarsening statistics but not pointwise accuracy. A hybrid strategy might also be considered, where efficient time steps based on operator splitting are taken during the spinodal phase or when other physics limits the time step size and our fully implicit time stepping strategy is used only during long ripening events. We consider the comparison of the efficiency of the approaches an open*

σ	time steps	ripening time	total PCG iterations
1e-4	70,517	13147	1,039,676
1e-5	202,549 (2.87)	9582	2,368,051 (2.27)
1e-6	618,431(3.06)	8695	5,205,739 (2.19)

Table 3.6 Performance of adaptive time stepping through a ripening event of the 1D Cahn-Hilliard model with $\epsilon = 0.18$ and $N = 128$ with Eyre's splitting (3.40). The numbers in brackets are ratios to quantities in the previous row. Compare to values in Table 3.3 for the fully implicit time discretization.

problem of interest. This is a question of accurate time stepping with lengthly solves versus time step size limited computations with fast solves. However, we remind the reader that the main advantage of the numerical framework described in this chapter is the generality of problems to which it can be easily applied rather than its potential as an efficient approach to any particular problem.

3.9.2 Asymptotics

We extend the asymptotic framework developed in section 3.4.2 to explore the solver efficiency and accuracy limitations of Eyre's splitting method.

3.9.2.1 Accuracy in Allen-Cahn equations

We highlight the accuracy limitations of Eyre's splitting in the simple, low order setting of the 1D Allen-Cahn problem (3.1). Consider the linear problem (3.17) in ripening states at which Theorem 3.4.1 applies. Applying a fully implicit BE time stepping method to this problem leads to the discrete problem

$$V^m = V^{m-1} + \delta t \mathcal{L} V^m \tag{3.42}$$

Consider now taking large time steps. Components of V in $\mathcal{V}^{\perp}$ are strongly reduced in a single time step. Components of V in $\mathcal{V}$ change slowly and represent the dynamics in the problem that need to be captured accurately during ripening. This characterizes the 1D Allen-Cahn equations as an *extremely* stiff problem, in the sense that it is even more stiff than the second order parabolic part acting on $\mathcal{V}^{\perp}$. For that reason, we strongly advocate the use of L-stable [88] time stepping schemes on this problem rather than weakly stable schemes such as trapezoidal rule or implicit midpoint rule. To consider the accuracy of the method, we make the standard ansatz

$$V^m = G^m \phi_j(x) \tag{3.43}$$

where G is a constant to be determined and recall that ϕ_j is an eigenfunction of $\mathcal{L}$ with small eigenvalue λ_j. Inserting (3.43) into (3.42) leads to the expected

$$G = \frac{1}{1 - \delta t \lambda_j} \approx 1 + \delta t \lambda_j + \delta t^2 \lambda_j^2 + \cdots \tag{3.44}$$

where we have expanded the expression in terms of small δt. Comparing to the exact $G = e^{\delta t \lambda_j}$ we see the term above has an error of $\frac{1}{2}\delta t^2 \lambda_j^2$, again as expected. We presented the standard analysis above to be able to compare it to the result from Eyre's method. Linearizing Eyre's method in this case leads to

$$V^m = V^{m-1} + \delta t(\mathcal{L} - I)V^m + \delta t V^{m-1}.$$

Here, it is seen why the Allen-Cahn equation is used for this discussion, since Eyre's splitting preserves the eigen-structure of $\mathcal{V}$ in this case. With the same ansatz (3.43) we obtain

$$G = \frac{1+\delta t}{1+\delta t - \delta t \lambda_j} \approx 1 + \delta t \lambda_j + \delta t^2(\lambda_j^2 - \lambda_j) + \cdots \tag{3.45}$$

in this case. Here the dominant error is $\lambda_j \delta t^2$, which is *exponentially larger* than the error in the fully implicit case. Another way to view this accuracy loss is that for the fully implicit method, if $\delta t \lambda_j$ is small, the scheme is "accurate" but for the Eyre's case, δt must be small when (3.45) is considered. Thus, exponentially smaller time steps must be taken with Eyre's scheme than with a fully implicit method. In this context, higher order splitting methods of Eyre's type based on standard implicit-explicit (IMEX) splitting do not solve this accuracy issue. While they are formally higher order accurate, they require time steps unreasonably small before they begin to be accurate, just as the lowest order scheme described above. We conjecture that this is true for all such IMEX approaches. For example, the "G" value for SBDF2 [102] is

$$1 + \delta t \lambda_j + \frac{1}{2}\delta t^2 \lambda_j^2 + \frac{\delta t^3}{6}(3\lambda_j^3 - 2\lambda_j^2) + \cdots$$

and for implicit-explicit midpoint rule [103]

$$1 + \delta t \lambda_j + \frac{1}{2}\delta t^2 \lambda_j^2 + \frac{\delta t^3}{6}(3\lambda_j^3/2 - 3\lambda_j^2/2) + \cdots$$

3.9.2.2 Condition number of solver for Eyre's method for Cahn-Hilliard problems

Here the relevant eigenvalue problem is

$$(I + \delta t D(\mathcal{L} - I))\psi = \sigma[I + \delta t D(\mathcal{L} - I + 3(u^2 - 1))]\psi \tag{3.46}$$

where I is the identity operator and D is the second derivative operator as before. The fundamental difference to the analysis in section 3.4.2.2 is that the operator multiplied by δt on the left is now $\mathcal{L} - I$ (which does not have small eigenvalues) rather than $\mathcal{L}$ (which does). This indicates there are not $O(1/\delta t)$ eigenvalues to this preconditioned system as $\delta t \to \infty$ but rather they remain $O(1)$ in this limit as observed computationally above. Formally considering (3.46) as $\delta t \to \infty$ leads to

$$(\mathcal{L} - I)\psi = \sigma(\mathcal{L} - I + 3(u^2 - 1))\psi \tag{3.47}$$

where it can be justified dropping the integration constant for $\sigma \neq 1$ at leading order due to the structure of $u^2 - 1$ at ripening states. The operators on either side above are negative definite and symmetric, so we obtain upper bounds for the smallest eigenvalues when we take $\psi = \phi_j$ (the eigenfunctions of $\mathcal{L}$ with small eigenvalues so $\mathcal{L}\psi_j =_e 0$) in the expression above and then take the inner product with ϕ_j. This gives asymptotically

$$\sigma_{\min} \leq \frac{1}{1 + 1/\beta_{AC}} \approx 0.294$$

where β_{AC} is the value (3.23) found previously for the relevant integral. We have not given convincing evidence that this should be the minimum eigenvalue. Considering (3.47) heuristically, the eigenfunction ψ for the minimum eigenvalue σ should minimize the size of $(\mathcal{L} - I)\psi$ and maximize the product with $1 - u^2$, which elements of $\mathcal{V}$ do. In addition, the value does correspond to that observed computationally.

3.10 Conclusions

We have presented a new approach to the computational approximation of energy gradient flows from material science models such as Allen-Cahn, Cahn-Hilliard and higher order and vector variants. The approach allows accurate time stepping with large time steps when the evolution is slow during ripening events. Some evidence is given that approaches based on Eyre's operator splitting require much smaller time steps to maintain accuracy in these settings. Fully implicit time stepping with matrix-free Newton steps solved with the CG method are proposed in this work. An efficient preconditioner is identified. Computational evidence and formal asymptotics show that the solver has mild computational increase as ϵ is decreased and time step δt is increased. The approach is shown to work well on a number of problems in a general class and allows for an efficient GPU implementation. It is possible that the computational approach may also be useful for other models such as those from the study of liquid crystals or epitaxial growth [107, 108] that share the same structure.

There are several avenues of study for theoreticians opened in this work. Arguments made using formal asymptotics backed by computational evidence in 1D could be made rigorous and extended to higher dimensions. The authors believe this is more than a technical exercise, but that real insight into the efficiency of different computational approaches can be obtained

from such a study. In addition, there is the conjecture in section 3.4.2.2 on the rank of a modified distance matrix.

There have been many computational approaches to the Cahn-Hilliard equation, which is used widely in materials science studies and as a computational approximation to moving interface problems. In this well-developed computational field, a test suite of benchmark problems is clearly needed. A review of the methods in the literature, including a comparison on the benchmark problems, would be a valuable contribution.

Chapter 4

Exponential Integrator

4.1 General Exponential Integrator

A significant challenge in simulating the Functionalized Cahn-Hilliard equation is finding the proper balance between time accuracy and reasonably fast computation times. Eyre's method uses a convex splitting of the equation to guarantee energy decay for any size timestep [46, 34], but the time accuracy suffers [109]. Alternatively, a fully implicit preconditioned conjugate gradient method gives the proper time evolution and time accuracy, but the Newton solve is severely restricted [109], resulting in a method that is too slow to reasonably capture the long-time dynamics of the solution. Our goal is to develop a method that is both time accurate and is computationally efficient enough to describe fully relaxed solutions to the FCH equation.

The exponential integrator method can provide the performance we desire under certain limitations, and it is also computationally efficient. The limitations are much less restrictive than the fully implicit method, and has much greater time accuracy than the convex splitting method. A review of the literature for exponential integrator methods is in Section 1.8.2. The work in this chapter follows the the papers published for ODEs by Du and Zhu under the name exponential time differencing (ETD) [66, 67], and by Cox and Matthews for stiff systems [65]. In this chapter we will present the ETD method for the CH and FCH equations, along with some variations in the time stepping to obtain greater stability and increased time

accuracy.

We desire to solve the general PDE defined by

$$u_t = F(u).$$

We first separate $F(u)$ into linear and non-linear parts,

$$u_t = \mathcal{L}u + \mathcal{N}(u). \tag{4.1}$$

By moving the linear portion to the left hand side and multiplying the equation by $e^{-t\mathcal{L}}$, we can obtain a total time derivative.

$$\begin{aligned} u_t - \mathcal{L}u &= \mathcal{N}(u) \\ e^{-t\mathcal{L}}u_t - e^{-t\mathcal{L}}\mathcal{L}u &= e^{-t\mathcal{L}}\mathcal{N}(u) \\ \left(e^{-t\mathcal{L}}u\right)_t &= e^{-t\mathcal{L}}\mathcal{N}(u). \end{aligned}$$

Integration in time gives the general exponential integrator formula,

$$u(t_{n+1}) = e^{\delta t\mathcal{L}}u(t_n) + e^{\delta t\mathcal{L}}\int_0^{\delta t} e^{-s\mathcal{L}}\mathcal{N}(u(s+t_n))\,ds \tag{4.2}$$

Up to this point, the derivation is exact, but we must now make approximations because we do not know enough about $\mathcal{N}(u(t))$ to compute the integral. The first approach is to obtain an explicit scheme that is first order in time by taking $\mathcal{N}(u(t)) \approx \mathcal{N}(u(t_n))$ over

the interval of integration. We then compute the exponential integral by hand to obtain

$$u_{n+1} = e^{\delta t \mathcal{L}} u_n + \frac{e^{\delta t \mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(u_n), \tag{4.3}$$

where the notation $u_n = u(t_n)$ is used here and in subsequent sections. Schemes that are higher order in time will be created by improving upon the approximation to $\mathcal{N}(u(t))$.

We use a Fourier spectral method to compute solutions to the CH and FCH equations (1.3 and 1.10). The spectral method comes with both advantages and disadvantages. The primary advantage is that the operators can be computed quickly and efficiently by hopping between the spectral and physical domains. This reduces nearly all of the calculations to element-wise operations on the arrays which can be computed extremely efficiently in parallel on a graphics processing unit (GPU). Further, solutions to the FCH equation develop layers with slope on the order of ε^{-1} [28], and the spectral method only needs a few grid points to capture such layers. The primary disadvantage of using a spectral method is that we are limited to periodic boundary conditions on rectangular domains. We expect to employ a finite element or discontinuous galerkin method when we need to change the domain and boundary conditions.

4.2 Stability and Linearization of the CH Equation

The most straight forward way to split the CH equation into the proper form (4.1) is to define

$$\mathcal{L}u = -\varepsilon^2\Delta^2 u - \Delta u$$
$$\mathcal{N}(u) = \Delta\left(u^3\right).$$

Unfortunately, this splitting is not stable with respect to inversion of $\mathcal{L}$. This can be seen by looking at the one-dimensional form of the operator in the frequency representation, where k is the wavenumber,

$$\mathcal{L} = -\varepsilon^2\left(-k^2\right)^2 - \left(-k^2\right) = -\varepsilon^2 k^4 + k^2. \tag{4.4}$$

Ideally, the eigenvalues of the operator would be negative for all values of k or at least nonzero, but here we have zeros at $k = \varepsilon^{-\frac{1}{2}}$. There are a couple of different ways to rectify this problem.

One way to stabilize the linear operator is to add and subtract a term in the equation that makes the operator non-positive in the frequency representation [56]. This gives,

$$u_t = -\varepsilon^2\Delta^2 u + \Delta\left(u^3 - u\right) + \beta\Delta u - \beta\Delta u, \tag{4.5}$$

with the splitting,

$$\mathcal{L}u = -\varepsilon^2\Delta^2 u + \beta\Delta u$$
$$\mathcal{N}(u) = \Delta\left(u^3 - (1+\beta)u\right).$$

In the frequency representation,

$$\mathcal{L} = -\varepsilon^2 k^4 - \beta k^2, \tag{4.6}$$

so any $\beta \geq 1$ gives the desired property. Through numerical simulations, we concur with Shen and Yang's suggestion that $\beta = 2$ is the ideal choice. Unfortunately, the addition of this term introduces additional error which is undesirable, but like the method described in Chapter 2, it comes with a guarantee that the energy will never increase. Thus, this method is ideal for simulations where dynamics are less important that the final geometry or solution. This decrease in accuracy is evident in Table 4.1, and it becomes more significant for the Functionalized Cahn-Hilliard equation.

A more accurate way to obtain stability under inversion is to linearize the problem using information from the physics of the solution. Solutions to the CH equation rapidly form domains where $u(x) = \pm 1$, so we choose to linearize about the background state $u(x) = -1$. We take $v = u + 1$ and substitute into equation 1.3 which gives

$$v_t = -\varepsilon^2\Delta^2 v + \Delta\left(2v - 3v^2 + v^3\right). \tag{4.7}$$

δt	Simple	Shen	Physics-based ETD
1e-3	1.3741	9.7420e-2	7.2249e-2
1e-4	4.8206e-4	4.5247e-3	2.5333e-3
1e-5	6.8251e-5	3.3740e-4	1.3448e-4
1e-6	1.0790e-5	2.3092e-5	6.5339e-6

Table 4.1 Comparison of linearization choices for the CH equation. Error of the final solution is given for a simulation with fixed time step δt. For the simple linearization with $\delta t = 10^{-3}$, the solution became unstable and developed spurious oscillations.

Taking the linear and nonlinear parts gives the operators

$$\mathcal{L}v = -\varepsilon^2 \Delta^2 v + 2\Delta v$$
$$\mathcal{N}(u) = \Delta\left(v^3 - 3v^2\right),$$

where the frequency representation of $\mathcal{L}$ is strictly negative for all values of k except $k = 0$, as desired.

With this formulation of the CH equation, we now apply the first order exponential integrator (4.3). We note that with this linearization, the first order method is unconditionally stable with respect to the time step size. Unfortunately, it suffers from a loss of convergence order for time steps larger than $\delta t \gtrsim 10^{-4}$.

4.3 Linearization of the FCH Equation

To effectively linearize the FCH equation, we use information from the physics of the solution. Solutions to the FCH equation rapidly form domains where $u(x) = -1$ is the dominant, background phase. We choose to linearize about the background state, $u(x) = b = -1$, by

taking $v = u - b$ and substituting into equation 1.10 which gives

$$\begin{aligned} v_t &= \Delta\left[\left(\varepsilon^2\Delta - W''(v+b) + \varepsilon\eta_1\right)\left(\varepsilon^2\Delta v - W'(v+b)\right) + \varepsilon(\eta_1-\eta_2)W'(v+b)\right] \\ &= \Delta\left[\left(\varepsilon^2\Delta - W''(b) - W'''(b)v - Q_2(v) + \varepsilon\eta_1\right)\left(\varepsilon^2\Delta v - W'(b) - W''(b)v - Q_1(v)\right)\right. \\ &\qquad \left. + \varepsilon(\eta_1-\eta_2)\left(W'(b) + W''(b)v + Q_1(v)\right)\right], \end{aligned}$$

where $Q_1(v)$ and $Q_2(v)$ contain the terms that are quadratic and higher order in v. For our typical potential well, $W_\tau(u) = \frac{1}{2}(u+1)^2\left(\frac{1}{2}(u-1)^2 - \frac{\tau}{3}(u-2)\right)$, we have $Q_1(v) = \frac{1}{2}(6b+\tau)v^2 + v^3$ and $Q_2(v) = 3v^2$. We note that $W'(b) = 0$, and separate the linear and nonlinear parts to obtain the operators

$$\begin{aligned} \mathcal{L}v &= \Delta\left[\left(\varepsilon^2\Delta - W''(b) + \varepsilon\eta_1\right)\left(\varepsilon^2\Delta - W''(b)\right) + \varepsilon(\eta_1-\eta_2)W''(b)\right]v \\ \mathcal{N}(u) &= \Delta\left[-\left(\varepsilon^2\Delta - W''(b) + \varepsilon\eta_1\right)Q_1(v) + \left(-W'''(b) - Q_2(v)\right)\left(\varepsilon^2\Delta v - W''(b)v - Q_1(v)\right)\right. \\ &\qquad \left. + \varepsilon(\eta_1-\eta_2)Q_1(v)\right]. \end{aligned}$$

This linearization is ideal because the frequency representation of $\mathcal{L}$ is strictly negative for all values of k except $k = 0$, making the operator stable under inversion for $\eta_1 \geq \eta_2$. Numerically, we observe stability as long as η_1 and η_2 are of the same order.

With this formulation of the FCH equation, we now apply the first order exponential integrator (4.3). We note that with this linearization, the first order method is unconditionally stable with respect to the time step size, and now extend this linearization to higher-order in time methods for solving the FCH equation.

4.4 Computing Exponential Terms

One way to improve time accuracy and obtain higher-order time-stepping is to expand $\mathcal{N}(v(t))$ as a Taylor series in time. Over the integral of integration, we take the expansion about t_n and truncate at the order of accuracy that we desire,

$$\begin{aligned}\mathcal{N}(v(t)) &= \mathcal{N}(v_n) + (t-t_n)\, J_{\mathcal{N}}\left(\frac{dv}{dt}\right)\Big|_{t_n} + O\left(\delta t^2\right) \\ &= \mathcal{N}(v_n) + (t-t_n)\, J_{\mathcal{N}}\left(\mathcal{L}v_n + \mathcal{N}(v_n)\right) + O\left(\delta t^2\right). \end{aligned} \tag{4.8}$$

$\frac{d}{dt}\mathcal{N}(v) = J_{\mathcal{N}}\left(\frac{dv}{dt}\right)$ is calculated by hand, noting that all the time dependence is in $v(x,t)$. We derive the second order method by first substituting the approximation into equation 4.2,

$$v_{n+1} = e^{\delta t\mathcal{L}} v_n + e^{\delta t\mathcal{L}} \int_0^{\delta t} e^{-s\mathcal{L}} \left[\mathcal{N}(v_n) + sJ_{\mathcal{N}}\left(\mathcal{L}v_n + \mathcal{N}(v_n)\right)\right] ds.$$

After pulling out the portions that are constant in time,

$$v_{n+1} = e^{\delta t\mathcal{L}} v_n + e^{\delta t\mathcal{L}} \left[\mathcal{N}(v_n) \int_0^{\delta t} e^{-s\mathcal{L}} ds + J_{\mathcal{N}}\left(\mathcal{L}v_n + \mathcal{N}(v_n)\right) \int_0^{\delta t} s e^{-s\mathcal{L}} ds\right],$$

we integrate the exponentials by hand to obtain a second-order in time method,

$$v_{n+1} = e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(v_n) + \frac{e^{\delta t\mathcal{L}} - 1 - \delta t\mathcal{L}}{\mathcal{L}^2} J_{\mathcal{N}}\left(\mathcal{L}v_n + \mathcal{N}(v_n)\right). \tag{4.9}$$

The exponential coefficients are universal for high order ETD methods, and they are particularly difficult to calculate numerically with reasonable accuracy. The exponential

coefficients are defined as:

$$\phi_0(z) = e^z \quad \phi_1(z) = \frac{e^z - 1}{z} \quad \phi_2(z) = \frac{e^z - 1 - z}{z^2} \quad \dots \quad \phi_{l+1}(z) = \frac{\phi_l(z) - \frac{1}{l!}}{z}. \quad (4.10)$$

In our implementation of ETD methods, $z = \delta t \mathcal{L}$, and we always compute these terms in the frequency representation. Using this notation, equations 4.3 and 4.9 can be written as,

$$v_{n+1} = \phi_0(\delta t \mathcal{L}) v_n + \delta t \ \phi_1(\delta t \mathcal{L}) \mathcal{N}(v_n)$$
$$v_{n+1} = \phi_0(\delta t \mathcal{L}) v_n + \delta t \ \phi_1(\delta t \mathcal{L}) \mathcal{N}(v_n) + \delta t^2 \ \phi_2(\delta t \mathcal{L}) J_{\mathcal{N}}(\mathcal{L} v_n + \mathcal{N}(v_n)),$$

respectively. In Sections 4.5 and 4.6, we introduce methods that are higher order in time which use the higher order ϕ-functions. Unfortunately, computing $\phi_1(z)$ directly encounters significant errors when z is small, i.e. both δt and k are small. This comes from cancellation errors when rounding off values at machine precision, and can lead to errors that are orders of magnitude larger than the actual value [62, 110, 69]. These errors become increasingly problematic for $\phi_l(x)$ with larger values of l.

To eliminate these errors, we use the Taylor expansion of $\phi_1(z)$ about $z = 0$, and include as many terms as necessary to obtain a desired tolerance. We find that direct computation of $\phi_l(z)$ is sufficient if $z > \eta^{\frac{1}{l+1}}$, where η is machine precision (typically 10^{-16}). In the spectral setting, $\mathcal{L}$ is small for wave numbers, k, near zero, but is typically $O(1)$ or larger. Thus, small z depends on the size of δt which can be as small as 10^{-12} in our three dimensional calculations. When computing $\phi_1(z)$ and $\delta t \leq 10^{-7}$, we divide the $\phi_1(z)$ operator into two portions: large k values are computed directly, and small k values are computed using a Taylor expansion out to a fixed number of terms that have been estimated *a priori*. In three

dimensional calculations, the cutoff for which the expansion is necessary is given by $\mathcal{L} < \frac{\sqrt{\eta}}{\delta t}$, where the array $\mathcal{L}$ was computed once at the beginning of the simulation. The calculation is performed similarly when approximating the higher order ϕ-functions.

An alternate method for stabilizing the calculation of ϕ-functions was presented by Kassam and Trefethen [69]. It involves calculating the value of an integral in the complex plane. When $\mathcal{L}$ is real, as the CH and FCH equations, the integration can be performed around any contour Γ that encloses the eigenvalues of the operator that are small in modulus. Kassam uses circles and 32 or 64 evaluation points with the trapezoidal rule to obtain stability and accuracy.

Similar work by Du and Zhu suggests that only two points are necessary for up to the fourth-order Runge-Kutta scheme [67]. It seems reasonable that only a few points are necessary to provide stability, but we find it difficult to believe that two points is sufficient to see higher order accuracy in time, and the paper gave no results higher than second order.

4.5 Runge-Kutta Methods

One way to improve the time accuracy of the exponential integrator is through the use of Runge-Kutta (RK) type methods. This is accomplished by computing approximations to $\mathcal{N}(v(t))$ using multiple stages over the interval $[t_n, t_{n+1}]$. For the second order RK method, we take an explicit exponential integrator step

$$a_n = e^{\delta t \mathcal{L}} v_n + \frac{e^{\delta t \mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(v_n),$$

then make the approximation

$$\mathcal{N}\left(v\left(t\right)\right) = \mathcal{N}\left(v_n\right) + \left(t - t_n\right)\left(\mathcal{N}\left(a_n\right) - \mathcal{N}\left(v_n\right)\right)/\delta t + \mathcal{O}\left(\delta t^2\right).$$

This approximation gives the ETDRK2 method described in [65, 66].

Denoting the numerical approximation to $v\left(t_n\right)$ as v_n, the second order RK method is

$$\begin{aligned} a_n &= e^{\delta t \mathcal{L}} v_n + \frac{e^{\delta t \mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}\left(v_n\right) \\ v_{n+1} &= a_n + \frac{e^{\delta t \mathcal{L}} - 1 - \delta t \mathcal{L}}{\delta t \mathcal{L}^2}\left(\mathcal{N}\left(a_n\right) - \mathcal{N}\left(v_n\right)\right). \end{aligned} \tag{4.11}$$

A similar construction gives a third-order exponential integrator RK method,

$$\begin{aligned} a_n &= e^{\frac{\delta t}{2}\mathcal{L}} v_n + \frac{e^{\frac{\delta t}{2}\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}\left(v_n\right) \\ b_n &= e^{\delta t \mathcal{L}} v_n + \frac{e^{\delta t \mathcal{L}} - 1}{\mathcal{L}}\left(2\mathcal{N}\left(a_n\right) - \mathcal{N}\left(v_n\right)\right) \\ v_{n+1} &= e^{\delta t \mathcal{L}} v_n + \delta t^{-2}\mathcal{L}^{-3}\left\{\left[-4 - \delta t\mathcal{L} + e^{\delta t \mathcal{L}}\left(4 - 3\delta t\mathcal{L} + \delta t^2 \mathcal{L}^2\right)\right]\mathcal{N}\left(v_n\right)\right. \\ &\qquad + 4\left[2 + \delta t\mathcal{L} + e^{\delta t \mathcal{L}}\left(-2 + \delta t\mathcal{L}\right)\right]\mathcal{N}\left(a_n\right) \\ &\qquad \left. + \left[-4 - 3\delta t\mathcal{L} - \delta t^2\mathcal{L}^2 + e^{\delta t \mathcal{L}}\left(4 - \delta t\mathcal{L}\right)\right]\mathcal{N}\left(b_n\right)\right\}. \end{aligned} \tag{4.12}$$

The standard fourth order Runge-Kutta method only attains third-order accuracy in time, but [65] gives a method with different coefficients that is fourth-order accurate for

exponential integrator schemes,

$$
\begin{aligned}
a_n &= e^{\frac{\delta t}{2}\mathcal{L}} v_n + \frac{e^{\frac{\delta t}{2}\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(v_n) \\
b_n &= e^{\frac{\delta t}{2}\mathcal{L}} v_n + \frac{e^{\frac{\delta t}{2}\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(a_n) \\
c_n &= e^{\frac{\delta t}{2}\mathcal{L}} a_n + \frac{e^{\frac{\delta t}{2}\mathcal{L}} - 1}{\mathcal{L}} \left(2\mathcal{N}(b_n) - \mathcal{N}(v_n)\right) \\
v_{n+1} &= e^{\delta t \mathcal{L}} v_n + \delta t^{-2} \mathcal{L}^{-3} \left\{ \left[-4 - \delta t \mathcal{L} + e^{\delta t \mathcal{L}} \left(4 - 3\delta t \mathcal{L} + \delta t^2 \mathcal{L}^2 \right) \right] \mathcal{N}(v_n) \right. \\
&\qquad + 2 \left[2 + \delta t \mathcal{L} + e^{\delta t \mathcal{L}} \left(-2 + \delta t \mathcal{L} \right) \right] \left(\mathcal{N}(a_n) + \mathcal{N}(b_n) \right) \\
&\qquad \left. + \left[-4 - 3\delta t \mathcal{L} - \delta t^2 \mathcal{L}^2 + e^{\delta t \mathcal{L}} \left(4 - \delta t \mathcal{L} \right) \right] \mathcal{N}(c_n) \right\}. \qquad (4.13)
\end{aligned}
$$

Since we are only using explicit methods, we require that these methods have sufficient numerical stability. Stability for the RK and other methods will be discussed in Section 4.8.

4.6 Taylor Methods

In Section 4.4, we derived a second-order, explicit ETD method based on a Taylor expansion of $\mathcal{N}(v)$ centered at t_n. Unfortunately, numerical simulations showed that this method (4.9) is unstable for large time-step sizes and attains second order convergence for only very small time-step sizes. This is due to approximating the function on the left hand side of the integration interval where the exponential multiplier is smallest (remember that $\mathcal{L} \leq 0$ for each k in the frequency representation). To solve this dilemma, we instead approximate

$\mathcal{N}(v(t))$ with an expansion about t_{n+1},

$$\begin{aligned}\mathcal{N}(v(t)) &= \mathcal{N}(v_{n+1}) + (t_{n+1}-t)\, J_{\mathcal{N}} \left(\frac{dv}{dt}\right)\Big|_{t_{n+1}} + O\left(\delta t^2\right) \\ &= \mathcal{N}(v_{n+1}) + (t_{n+1}-t)\, J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1})) + O\left(\delta t^2\right). \end{aligned} \tag{4.14}$$

Again substituting into equation 4.2 and extracting the multiplying constants from the integral, we have

$$\begin{aligned} v_{n+1} &= e^{\delta t\mathcal{L}} v_n + e^{\delta t\mathcal{L}} \int_0^{\delta t} e^{-s\mathcal{L}} \left[\mathcal{N}(v_{n+1}) + (\delta t - s)\, J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1}))\right] ds \\ &= e^{\delta t\mathcal{L}} v_n + e^{\delta t\mathcal{L}} \Bigg[\left\{\mathcal{N}(v_{n+1}) + \delta t J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1}))\right\} \int_0^{\delta t} e^{-s\mathcal{L}} ds \\ &\qquad - J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1})) \int_0^{\delta t} s e^{-s\mathcal{L}} ds \Bigg]. \end{aligned}$$

Computing the integrals by hand leaves an implicit numerical method,

$$\begin{aligned} v_{n+1} = e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} &\left\{\mathcal{N}(v_{n+1}) + \delta t J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1}))\right\} \\ &- \frac{e^{\delta t\mathcal{L}} - 1 - \delta t\mathcal{L}}{\mathcal{L}^2} J_{\mathcal{N}}(\mathcal{L}v_{n+1} + \mathcal{N}(v_{n+1})). \end{aligned}$$

Rather than solve this implicit method, we add an explicit stage to approximate v_{n+1},

$$\begin{aligned} a_n &= e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(v_n) \\ v_{n+1} &= e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} \left\{\mathcal{N}(a_n) - \delta t J_{\mathcal{N}}(\mathcal{L}a_n + \mathcal{N}(a_n))\right\} \\ &\qquad + \frac{e^{\delta t\mathcal{L}} - 1 - \delta t\mathcal{L}}{\mathcal{L}^2} J_{\mathcal{N}}(\mathcal{L}a_n + \mathcal{N}(a_n)). \end{aligned} \tag{4.15}$$

This method based on the Taylor expansion is not unconditionally stable with respect to time-step size, but it allows a much larger time step size than (4.9) and shows second order convergence (see Section 4.7). Further, since the second stage of equation 4.15 accepts an approximation for v_{n+1} and returns a better approximation for it, we can iterate on the second stage of the method to reduce the error of our solution. As the number of iterations increases, the error in the solution approaches the error of a fully-implicit second-order exponential integrator. In this way, the method behaves like a predictor-corrector type method.

By taking more terms in our initial Taylor expansion, a similar third order method is

$$\begin{aligned}
a_n &= e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} \mathcal{N}(v_n) \\
\dot{\mathcal{N}}(a_n) &= J_{\mathcal{N}}(\mathcal{L}a_n + \mathcal{N}(a_n)) \\
\ddot{\mathcal{N}}(a_n) &= H_{\mathcal{N}}\left([\mathcal{L}a_n + \mathcal{N}(a_n)]^2\right) + J_{\mathcal{N}}\left[\mathcal{L}(\mathcal{L}a_n + \mathcal{N}(a_n)) + J_{\mathcal{N}}(\mathcal{L}a_n + \mathcal{N}(a_n))\right] \\
v_{n+1} &= e^{\delta t\mathcal{L}} v_n + \frac{e^{\delta t\mathcal{L}} - 1}{\mathcal{L}} \left\{ \mathcal{N}(a_n) - \delta t\dot{\mathcal{N}}(a_n) + \frac{\delta t^2}{2}\ddot{\mathcal{N}}(a_n) \right\} \\
&\quad + \frac{e^{\delta t\mathcal{L}} - 1 - \delta t\mathcal{L}}{\mathcal{L}^2} \left\{ \dot{\mathcal{N}}(a_n) - \delta t\ddot{\mathcal{N}}(a_n) \right\} \\
&\quad + \frac{e^{\delta t\mathcal{L}} - 1 - \delta t\mathcal{L} - \frac{1}{2}\delta t^2\mathcal{L}^2}{\mathcal{L}^3} \ddot{\mathcal{N}}(a_n).
\end{aligned} \tag{4.16}$$

Again, each iteration of the second stage improves the solution by decreasing the error. Since the predictor stage is only first order, we require at least two iterations of the second stage to obtain a third order method. Further iterations drive the solution to the expected error of a fully-implicit third-order exponential integrator.

4.7 Convergence of ETD Methods

To test the order of convergence in time step size, and to compare the methods discussed in this chapter, we completed a refinement study. The initial condition was a solution to the CH equation that was far past the spinodal phase but still far from the steady state solution. Time evolution was computed for fixed $\delta t = 10^{-4}$ then halved and recomputed. We estimated the error for $\delta t = 10^{-4}$ as the distance between the solutions measured in the maximum norm. Using the solution with $\delta t = 5 \cdot 10^{-5}$, we again halved the time step to calculate the corresponding error. Figures 4.1 through 4.5 give the results of the refinement studies for each of the methods described above.

Figure 4.1 shows the results for the higher order ETD-Runge-Kutta methods (equations 4.11, 4.12, and 4.13) together with the general ETD (4.3) for comparison. Dashed lines with the proper order are included for reference. We note that for small enough time step size, each of the methods reaches its expected order of convergence. However, as the size of the time step approaches $\delta t = 10^{-4}$ the order of convergence begins to plateau. The reason for this behavior is discussed in Section 5.4.1.

Figure 4.2 gives the corresponding plot for the ETD-Taylor methods (equations 4.15 and 4.16). The methods reach their expected order of convergence for larger size time steps than the ETD-RK methods, but again they plateau near $\delta t = 10^{-4}$. Figures 4.3 and 4.4 show the effect of iterating on the second stage of the method for the ETD-T2 and ETD-T3 methods, respectively. These iterations reduce the error by as much as three orders of magnitude in the ETD-T2 method and five orders of magnitude in the ETD-T3 method.

Lastly, Figure 4.5 compares the error between the Runge-Kutta and Taylor methods for both the second and third order schemes. For the simple implementation, the Taylor

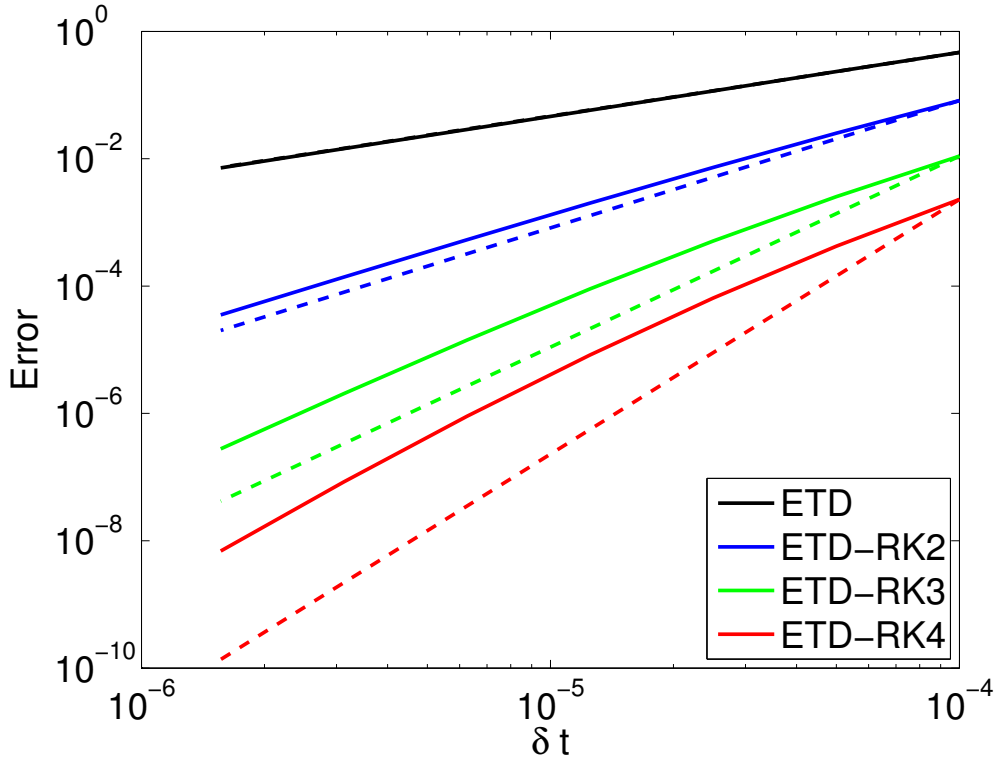


Figure 4.1 Timestep size refinement study for the ETD-RK methods given in Section 4.5. Dashed lines are included for comparison and have slope of two, three, and four respectively.

methods have slightly more error than the corresponding ETD-RK methods, but including several iterations makes the Taylor methods much more valuable.

4.8 Stability Analysis

In this section I compare the stability regions of the first through third order ETD methods described above (4.3, 4.11, 4.12, 4.15, 4.16). The stability region is the parameter region such that the magnitude of the amplification factor is less than or equal to one when it is applied to the model equation

$$u_t = \xi u + \lambda u, \tag{4.17}$$

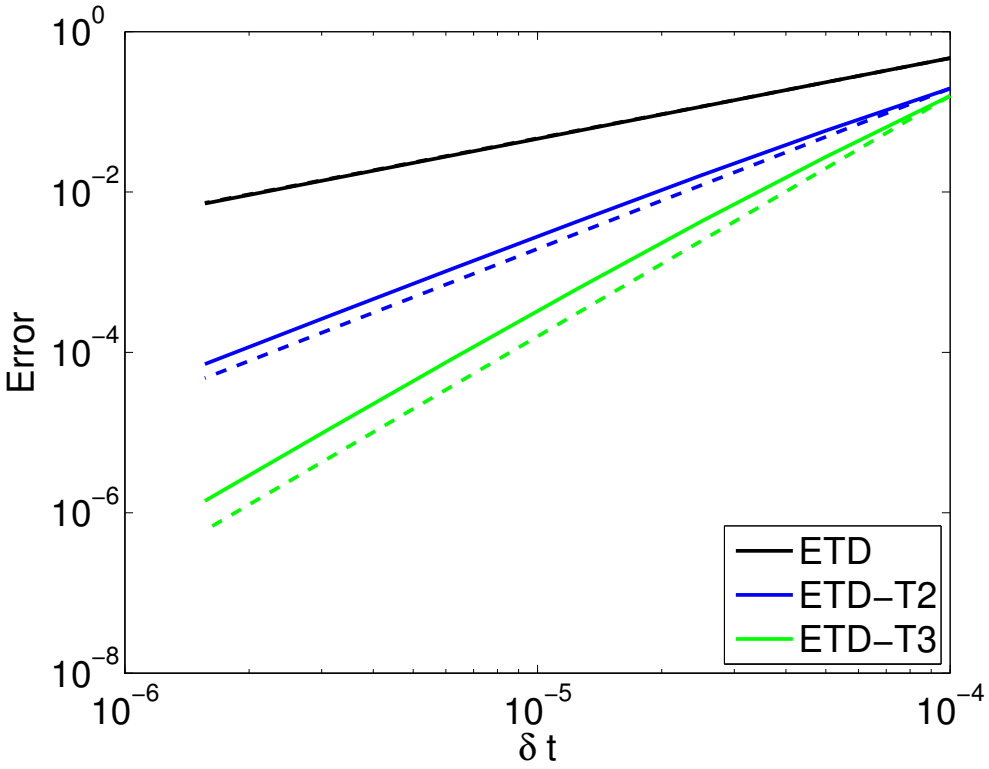


Figure 4.2 Timestep size refinement study for the ETD-Taylor methods given in Section 4.6. Dashed lines are included for comparison and have slope of two and three respectively.

where $\mathcal{L}u = \xi u$, and $\mathcal{N}(u) = \lambda u$. We assume ξ is a negative, real-valued constant and λ is complex. In Figures 4.6, we plot the regions for λ in the complex plane where $u_{n+1} \leq 1$ for one step with an initial condition $u_n = 1$. Similar plots and analysis for the ETD-RK methods can be found in [67] and [65].

As discussed in Section 4.6, the Taylor methods decrease error when the second stage is repeated, but unfortunately iterating on the corrector stage does not significantly increase the size of the stability region as shown in Figure 4.7.

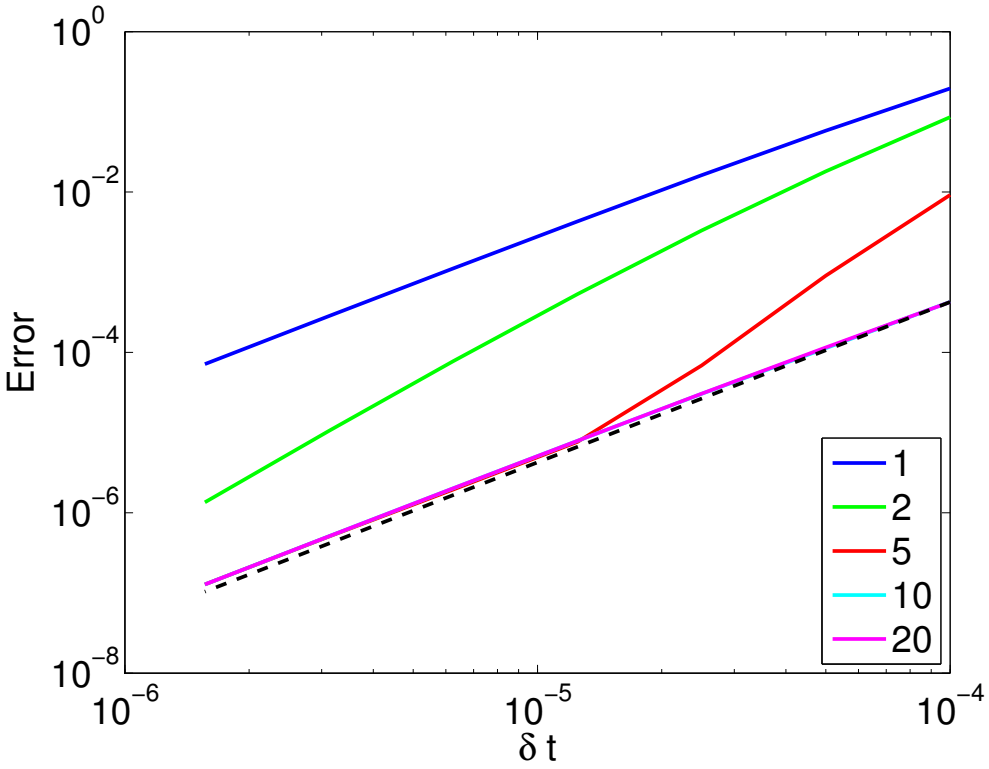


Figure 4.3 Timestep size refinement study for the ETD-T2 method with different numbers of iterations. The dashed line has slope two and is included for comparison.

4.8.1 Predictor/Corrector Analysis

To understand the limits on the stability region for the ETD-Taylor methods, we take the limit as $\delta t\xi \to 0$ and look at the ode,

$$u_t = \lambda u. \tag{4.18}$$

The backward Taylor series for this problem yields a single time step of the form

$$u_{n+1} = u_n + \left(z - \frac{z^2}{2!} + \frac{z^3}{3!} - \frac{z^4}{4!} + \cdots \right) u_{n+1}, \tag{4.19}$$

where we can truncate the expansion at the order we desire for the method.

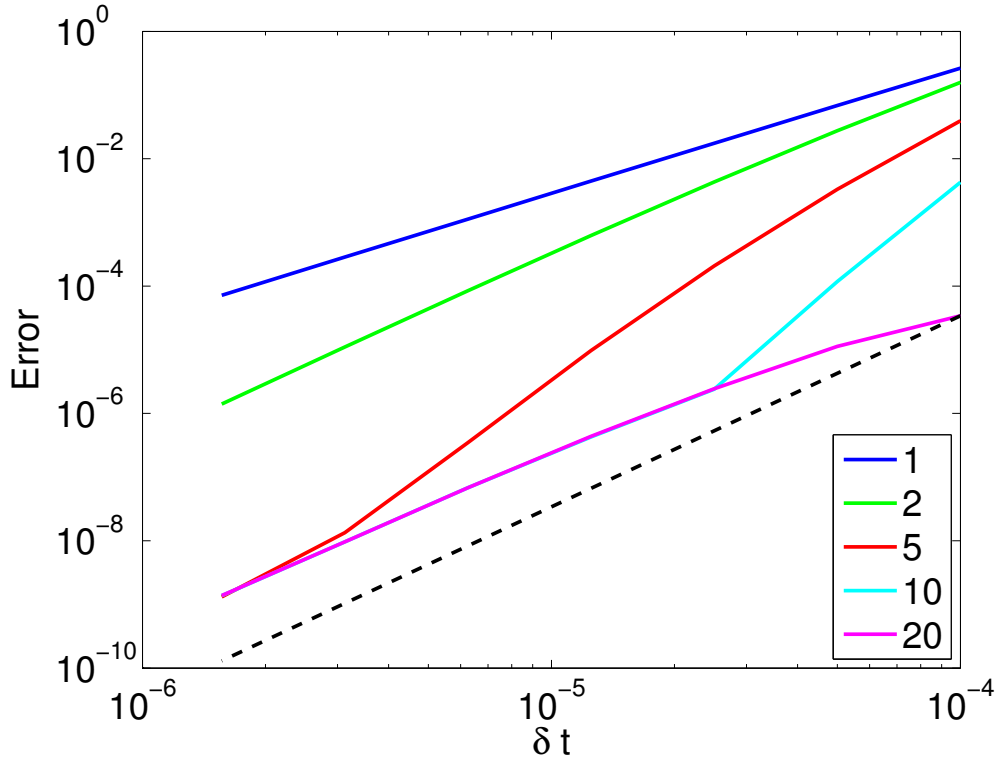


Figure 4.4 Timestep size refinement study for the ETD-T3 method with different numbers of iterations. The dashed line has slope three and is included for comparison.

Taken together with a predictor step, we have:

$$\textbf{Predictor:} \qquad u^{[0]} = p_0(z) \cdot u_n \tag{4.20a}$$

$$\textbf{Corrector:} \qquad u^{[k]} = u_n + p(z) \cdot u^{[k-1]}, \tag{4.20b}$$

with k as the counter for number of corrector iterations. To implement the second order ETD-T2 scheme (4.15) for the ode of interest (4.18), we use $p_0(z) = 1 + z$ in (4.20a) which yields the first order predictor, and set $p(z) = z - z^2/2$ in (4.20b) giving the second order corrector. Each iteration of (4.20b) will increase the order of the method, up to the order of accuracy of $p(z)$. Thus the second step of ETD-T3 must be performed at least twice to achieve solutions that are third order in time.

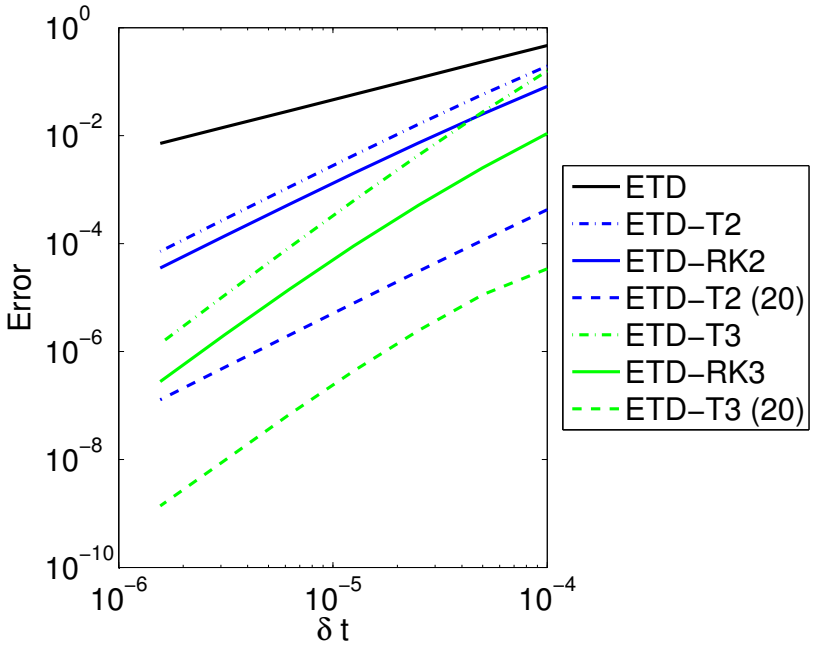


Figure 4.5 Comparison of the error for ETD-RK methods versus the ETD-Taylor methods with one and twenty iterations.

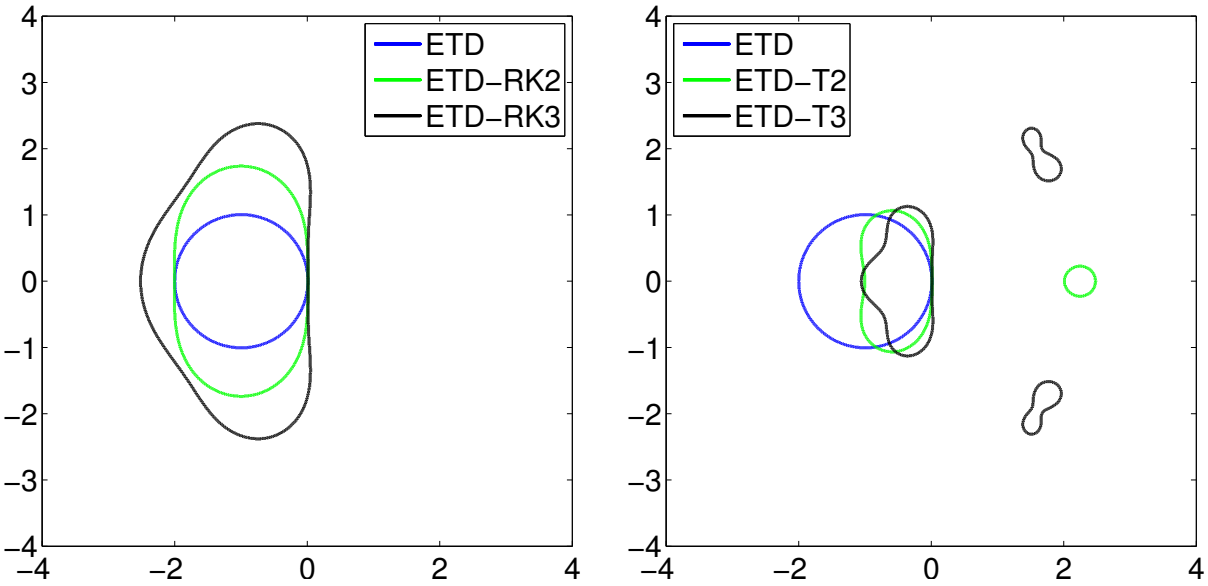
The exact solution to the linear difference equation (4.20a)-(4.20b) is given by,

$$u^{[k]} = \frac{u_n}{1-p(z)} + p(z)^k \left(p_0(z) - \frac{1}{1-p(z)} \right) u_n. \tag{4.21}$$

Provided $|p(z)| < 1$, the error converges to the same as a fully implicit scheme, but unfortunately this has a much smaller region of stability. For second and third order corrector steps, Figure 4.8 shows the regions where $|p(z)| < 1$ which match with the results of Figure 4.7 for small $\delta t \xi$.

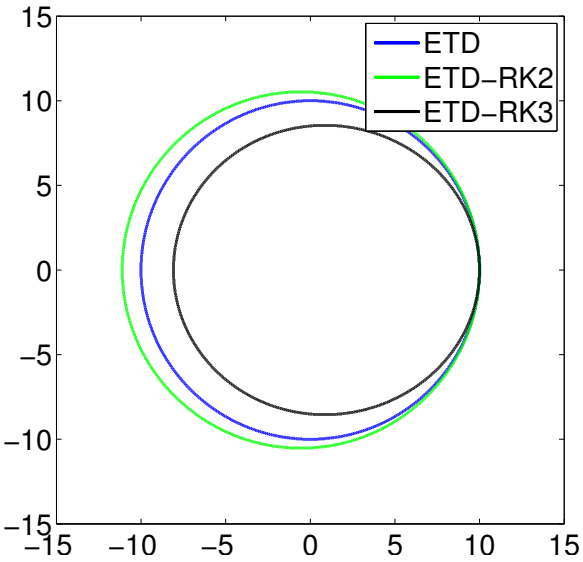
4.9 Implementation of ETD-RK2

To obtain larger-scale, three-dimensional solutions to the CH and FCH equations, we implemented the ETD-RK2 method for time stepping combined with a spectral solution in space.

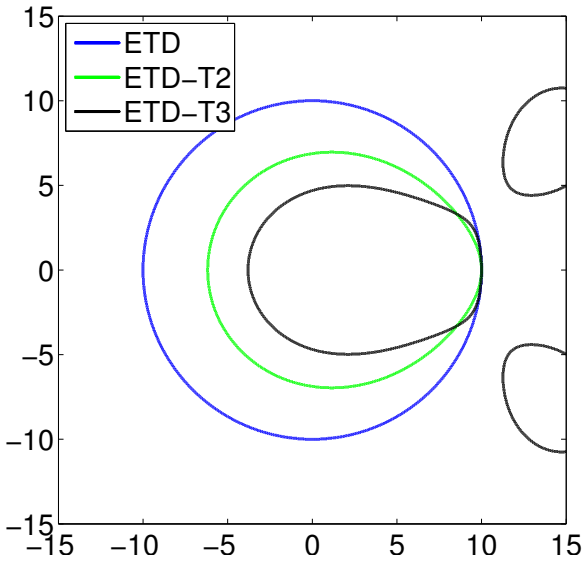


(a) Runge-Kutta $\delta t\xi = -0.01$

(b) Taylor $\delta t\xi = -0.01$

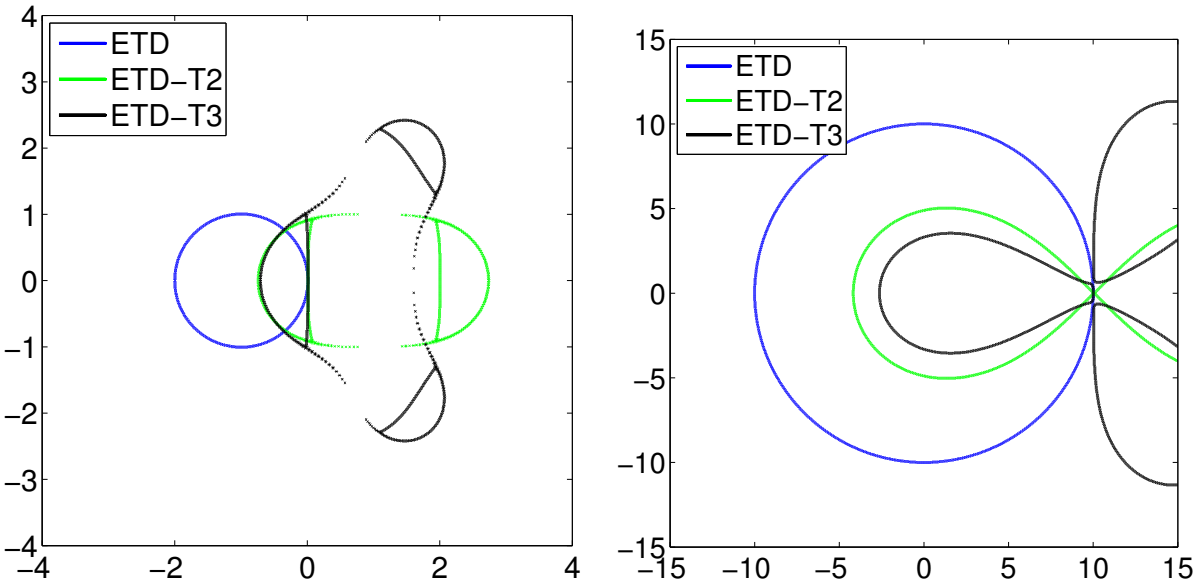


(c) Runge-Kutta $\delta t\xi = -10$



(d) Taylor $\delta t\xi = -10$

Figure 4.6 Stability regions for exponential time differencing methods up to third order in time. Regions are plotted in the $\delta t\lambda$ complex plane.



(a) $\delta t\xi = -0.01$ (b) $\delta t\xi = -10$

Figure 4.7 Stability regions for ETD-T2 and ETD-T3 methods with 100 iterations of the second step. Regions are plotted in the $\delta t\lambda$ complex plane.

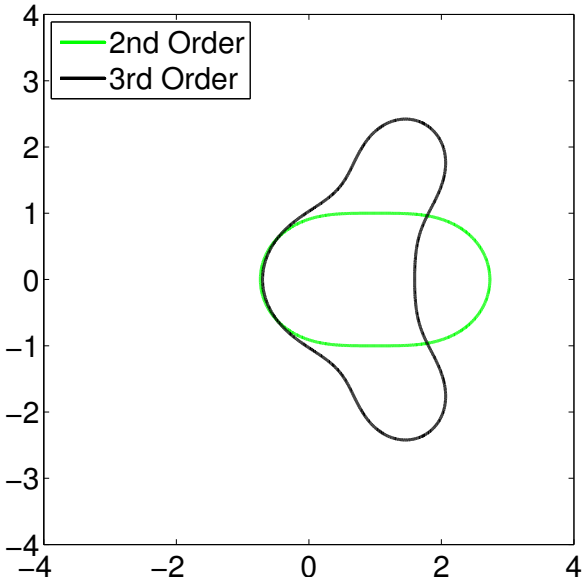


Figure 4.8 Contours where $p(z) = 1$, giving limits on the region of stability for ETD-T2 and ETD-T3. The regions are plotted in the $\delta t\lambda$ complex plane. Compare with Figure 4.7

Two properties make the spectral method ideal for this calculation: high spatial resolution captures solution fronts, and computing derivatives is fast and easy. Details for the general spectral method are given in Section 3.3. We first wrote the code in C++ and parallelized it using OpenMP. After we were satisfied with the stability and validity of the CPU version of the code, we wrote a corresponding code for a GPU using C++/CUDA. The Fourier transforms were computed using cuFFT, a package similar to FFTW adapted for GPUs. We used an adaptive time-stepping scheme based on error control, and a cubic domain with 128 grid points in each direction. The cubic domain is limited in the number of Fourier modes in each dimension by the size of the GPU's on-card memory. We performed our calculation on an Nvidia M1060 which has 4 gigabytes of memory leading to the 128 mode limitation.

4.9.1 Application to a GPU

When writing algorithms for a graphics processing unit there are three important hardware properties; massive parallelism, fixed memory limit, and asynchronous scheduling. Algorithms that are effective within these limitations can gain impressive speed as shown in the previous examples (Sections 2.9 and 3.6.1).

Modern GPUs typically have thousands of streaming multiprocessors that can handle thirty-two threads at a time. To be computationally effective, this impressive parallelism requires problems of sufficient size. If the computational requirements of the algorithm are too small, multiprocessors will sit idle and lead to a decrease efficiency. On the other hand, each GPU has limited on-card memory, so an effective algorithm will not require large amounts of storage. This on-card memory is limited to about 6GB of storage for current GPUs, and any memory transfer onto and off of the card is much slower than the speed of calculation, so it is effectively a hard limit on the amount of usable memory.

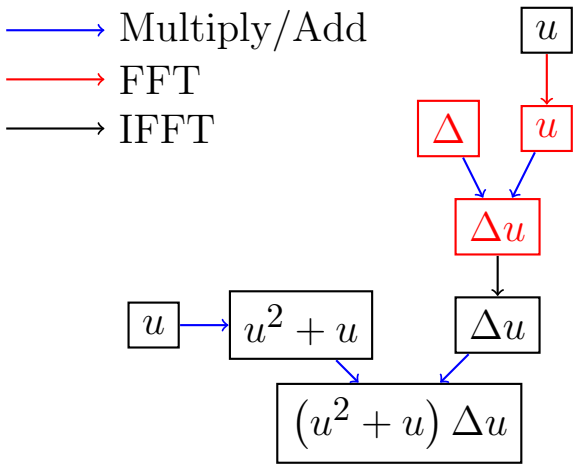


Figure 4.9 Diagram showing how to implement a spectral method on a GPU to avoid synchronization, minimize memory use, and fully utilize the parallelism of the GPU.

Finally, graphics processing units gain speed by hiding latency. This is done by scheduling with a first-come-first-served method for each of the streaming multiprocessors. Every time an algorithm requires a synchronization of the threads, the scheduler must clear out and restart, which slows down the computation. This is particularly relevant when computing with a spectral method because synchronization must occur when switching between regions of element-wise calculations and regions of Fourier transforms.

Figure 4.9 gives the process for computing $\left(u^2 + u\right) \Delta u$. This term does not appear in either the CH or FCH equations, but it gives a simple working example to better explain how the more complicated algorithms work. Beginning with two arrays, u in the spatial domain and Δ in the frequency domain, we alternate between FFT/IFFTs and element-wise operations. Note that by computing $u^2 + u$ at the last level we are able to perform the calculation requiring storage of only one additional array. Further, when the domain is relatively large, every streaming multiprocessor will be required giving full usage of the massive parallelism.

4.9.1.1 GPU Speedup

For the ETD-RK2 method, we conducted a speed test similar to Section 2.9. The test consisted of a very short simulation (approximately thirty time steps) computed with a single graphics processing unit compared against a fully parallelized code running on eight CPU processors. The hardware for the two codes was purchased at roughly the same time by the High Performance Computing Center at Michigan State University, so the comparison is fair. The same random initial data was used for each simulation, and the parameters were: $\varepsilon = 0.03$, $\eta_1 = 5.0$, $\eta_2 = 3.0$, and $\tau = 0.6$. We adjusted the number of grid points in the domain by powers of two in each dimension keeping the shape as close to cubic as possible.

The CPU simulations were run on two quad-core Intel Xeon E5620 processors, with each of the eight cores having 2.4 GHz of processing speed. The code was written in C++ and aggressively parallelized with OpenMP using the FFTW library for the Fourier transforms.

The GPU simulation was on an Nvidia Tesla C2075 which has 448 cores with 1.15 GHz processing speed. The code was written in C++ and CUDA using the cuFFT library for the Fourier transforms.

Figure 4.10 shows the number of times faster that the GPU calculation ran compared to the CPU calculation. As discussed before, the total computation speedup is lower than the speed up per time-step due to the slow transfer of data to and from the GPU card. As computation length increases the total computation speed up will approach the higher value. Further, the size of the problem is also important. As the number of grid points increases, so does the amount of computation between thread synchronizations on the GPU leading to greater efficiency. The number of grid points for this computation cannot be increased above $10^2 2$ due to the 6GB memory limit on the card.

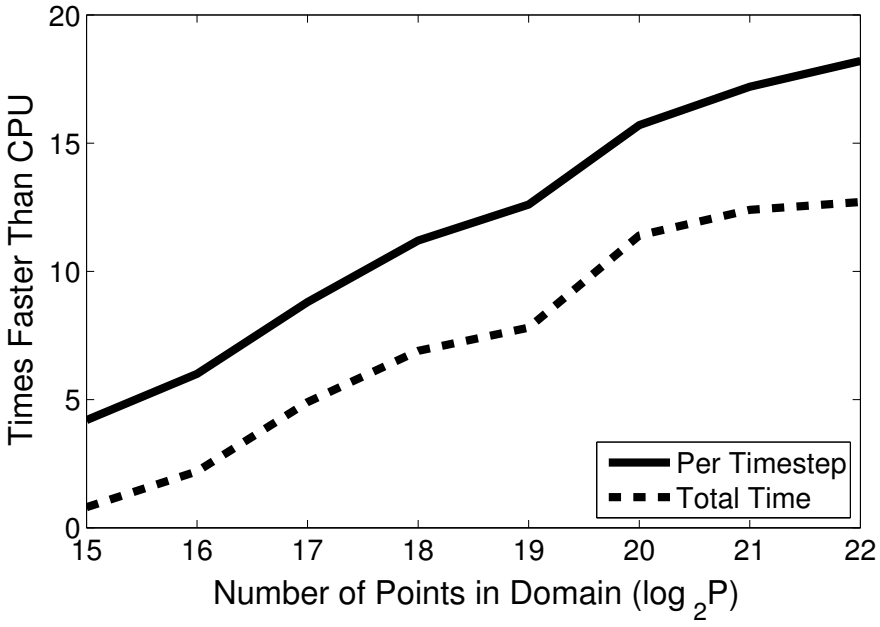


Figure 4.10 Number of times faster the GPU computation is over an eight-core, fully-parallelized, CPU computation.

An aspect of supercomputing centers that is gaining notoriety is the issue of power consumption [111]. For this speed comparison, the CPU calculation used eighty watts of energy per processor for a total energy consumption of 160 W, while the GPU used 225 Watts. Although the CPU used less power per unit time, the total power consumption was two to nine times less for the GPU than the CPUs, because the GPU completed the calculation many times faster. Energy efficiency is an additional advantage of GPU computing.

4.9.2 Adaptive Time-Stepping

In order to capture fast transient behaviors in the Cahn-Hilliard and Functionalized Cahn-Hilliard simulations, we implemented a second order adaptive time stepping scheme with the ETD-RK2 method. Unlike the pairs of methods described in Chapter 3, we do not have a corresponding implicit scheme with matching error terms. Instead we use the trapezoidal

method to compare against,

$$u_{n+1} = u_n + \frac{\delta t}{2}\left(F\left(u_{n+1}\right) + F\left(u_n\right)\right). \tag{4.22}$$

It would be pointless to solve the second order implicit equation, so we take the solution from ETD-RK2 at the new time, $u_{\text{ETD-RK2}}$, substitute it into equation 4.22, and explicitly compute an approximation to the solution from the trapezoidal method,

$$u_{\text{trap}} = u_n + \frac{\delta t}{2}\left(F\left(u_{\text{ETD-RK2}}\right) + F\left(u_n\right)\right). \tag{4.23}$$

Taking the difference of these two solutions at time t_{n+1}, we obtain an estimate, η, of the exact single-step error, η_e,

$$\eta_e \approx \eta := \frac{1}{2}\left\|u_{\text{ETD-RK2}} - u_{\text{trap}}\right\|. \tag{4.24}$$

This estimate is used to choose the size of the next time step, δt_{n+1}, with

$$\delta t_{n+1} = \delta t_n \max\left(0.8\sqrt[3]{\frac{\sigma}{\eta}}, 1.3\right),$$

where σ is the desired tolerance, and as before, 0.8 and 1.3 are safety factors.

The adaptive time-stepping is primarily important for the beginning stages of a simulation where the solution rapidly relaxes onto the slow manifolds of evolution. It also becomes important later in simulations where domains grow close together and rapid topological changes occur. At all times, we restrict our time-step to $\delta t \leq 10^{-4}$, for the reasons discussed in Sections 4.7 and 5.4.1.

4.10 Numerical Examples for FCH

As with the other methods, we we used the high-performance solver to investigate important geometries and structural evolutions that had connections to the analysis of the models. Using the medusa head problem in two dimensions, we show that the ETD-RK2 method gives proper convergence in time for important topological events. Second, we applied the three-dimensional GPU solver to a geometry with a hollow sphere interacting with hoop shaped pores. Third, we will show evolution of a punctured hollow sphere that has applications to vesicle membranes in biology.

4.10.1 Medusa Head in 2D

The medusa head problem forced us to make some important changes to our early choice of time-stepping method for the FCH equation. In Section 2.7, we initially presented the medusa head in three dimensions where we used the convex splitting method to compute time evolution. When we went back to show convergence in time for the precipitous drop in energy, we discovered that when we cut the time step in half, the event happened in approximately half the time. If this effect was due to compounding error that drove the system away from the unstable equilibrium, we would expect that halving the time step would lengthen the time before the event.

We chose this geometry as a check against the undesirable behavior shown by the convex splitting method. In Figure 4.11, we ran the simulation with the same initial condition and fixed time-steps of $2 \cdot 10^{-4}$, 10^{-4}, and $5 \cdot 10^{-5}$. The event did happen sooner for smaller time step sizes, but difference was very minor. This suggests that our time evolution is precise, and future work will address accuracy in time evolution compared against the asymptotic

analysis of the model. The calculation was done in MATLAB.

4.10.2 Punctured Hollow Spheres

An area of research in cell biological where the FCH model could prove useful is in the study of lipid membranes. Cells use vesicles (hollow spheres) composed of lipid bilayer membranes to transport nutrients and perform other vital functions. Recently, molecular dynamics simulations have been used to study formation of these membranes [112, 113]. Using our phase field model, we studied the time evolution of a vesicle that had been punctured by removing the section with azimuthal angle, ϕ, less than $\frac{\pi}{8}$, $\frac{\pi}{16}$, and $\frac{\pi}{32}$.

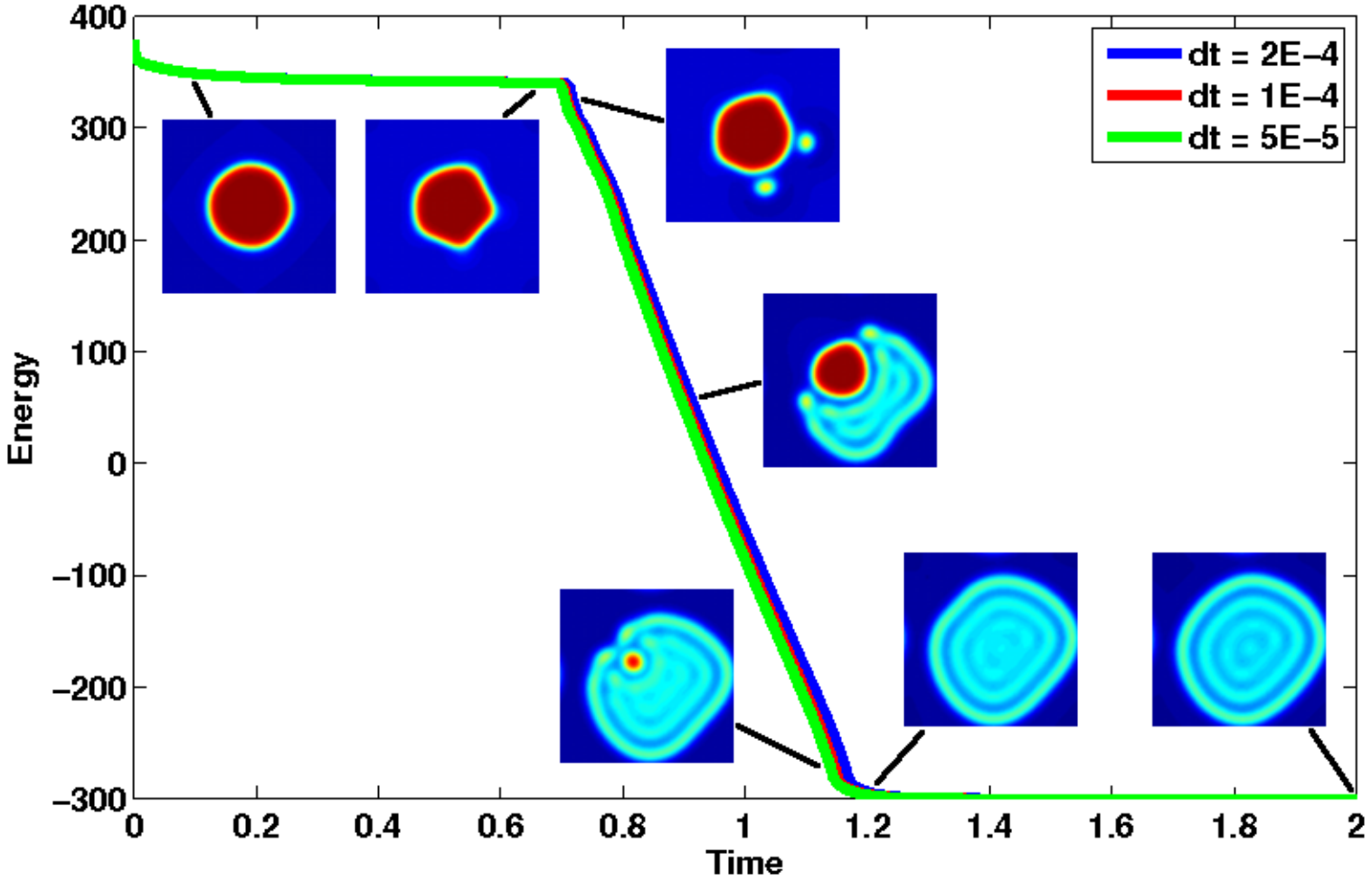


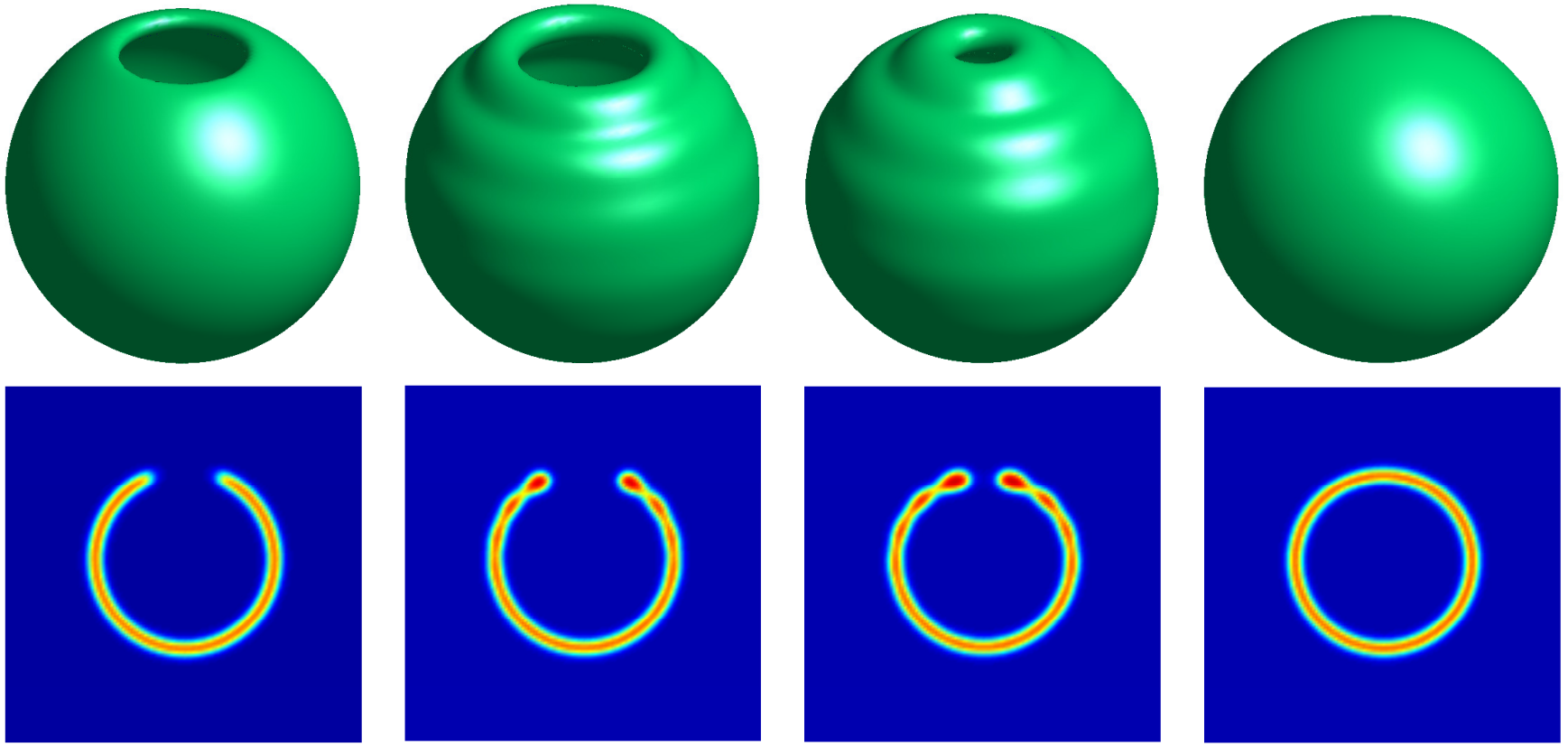
Figure 4.11 Time evolution of an unstable steady state solution to the FCH equation in two dimensions. Energy versus time is shown for three identical simulations with different fixed time step sizes.

Figure 4.12 shows the time evolution of the punctured vesicle with $\phi \leq \frac{\pi}{8}$ removed. Initially, the opening receded and reduced the line-energy by thickening the edge. After, the opening had stabilized, the hole slowly closed. The parameters for simulation were: $\varepsilon = 0.03$, $\eta_1 = 5$, $\eta_2 = 10$, $\tau = 0.2$, and the domain was $[-\frac{\pi}{2}, \frac{\pi}{2}]$ cubed. In non-dimensionalized time, the closing times for the three punctured spheres was $T = 10.9934$, $T = 0.9658$, and $T = 0.03217$ respectively, as measured by the sign change of the second derivative of the energy decay from negative to positive.

4.11 Conclusions

The exponential time differencing schemes provide a good balance of accuracy and speed when computing solutions to the Functionalized Cahn-Hilliard equation (1.10). The general ETD scheme can be extended to higher-order time-stepping schemes by approximating the nonlinear portion of the equation using both Runge-Kutta or Taylor expansions. Finding a good linearization of the right hand side of the equation provides dividends in both stability and accuracy of the calculation. The best linearization that we found was based on the physics of the model by computing the linearization about the dominant phase $u = -1$.

We studied the stability and accuracy for the higher-order ETD-RK and ETD-Taylor methods and found that in general the Runge-Kutta methods had better stability properties and accuracy for the simple schemes. However, when we treated the ETD-Taylor methods like predictor corrector schemes, it only took a few iterations to obtain a scheme hundreds of times more accurate that either the ETD-RK or simple ETD-Taylor methods. Higher-order ETD schemes rely on precise calculation of the ϕ-function to be stable and accurate.



$T = 0$ $T = 0.5$ $T = 10.0$ $T = 15.0$

Figure 4.12 Time evolution of a punctured vesicle.

Adaptive time-stepping based on error estimates improved the speed of the calculations for reasonable accuracy. We were able to use this combination of speed and accuracy to capture accurate evolution of physically and biologically relevant structures. Certainly, exponential time differencing scheme are adequate for simulation of the FCH equation.

Chapter 5

Comparison of Methods for the FCH Equation

To quantify the differences between the methods proposed in Chapters 2, 3, and 4, we performed an experiment on the punctured hollow sphere geometry from Section 4.10.2 using the three-dimensional GPU codes given in Appendix **??**. The initial condition used was a sphere with angle $\phi \leq \frac{\pi}{16}$ removed and evolved numerically until a stage after the hole had closed, but the sphere had not yet become smooth. This gives a smooth geometry on the manifold of evolution that will evolve quickly without any changes in topology. The parameters for the test were: $\varepsilon = 0.03$, $\eta_1 = 5\varepsilon = 0.15$, $\eta_2 = 10\varepsilon = 0.30$, and $\tau = 0.20$. We compare the accuracy, computation speed, and time step restrictions for the three methods computed on the GPU.

5.1 Description of Compared Schemes

The implementation of the convex splitting method applied the scheme from equation 2.5 with the splitting given in equations 2.12 and 2.13. The fixed-point iteration uses the operator defined by equation 2.31 with $c_1 = c_2 = 5.0$. The error from this method is expected to be $O(\delta t)$, and the adaptive time-stepping constant is set at $c_t = 0.16$ (see 2.41).

The implementation of the implicit method used the scheme described in Section 3.3.3

with the Jacobian matrix and associated preconditioned matrix given in equations 3.13 and 3.14. The error from this method is $O(\delta t)$, and the adaptive time stepping tolerance was set to $\sigma = 10^{-4}$.

The implementation of the exponential time differencing used the ETD-RK2 scheme (4.11), with the linearization from Section 4.3. The error from this method is $O\left(\delta t^2\right)$. We compute adaptive time stepping based on the difference between the solutions after both stages at time $t = t_{n+1}$. This gives a fast first-order adaptive time stepping that does not require any extra computation.

5.2 Complete Simulation

For the first comparison, we ran a simulation from $T = 0$ to $T = 2$ with each of the three methods. We ran the simulations using the fully parallelized and optimized GPU code, and we allowed completely adaptive time stepping. Results from the simulations are given in Table 5.1. To obtain error estimates, we compared against a "true" solution computed using the ETD-RK2 method with a fixed time step size of $\delta t = 10^{-6}$. We also computed the solution with a fixed time step size $\delta t = 10^{-5}$ so that we could be confident that this "true" solution was accurate enough (i.e. the error was at least an order of magnitude smaller than the comparison solutions).

The first major result was that the solution from the convex splitting method did not evolve enough to close the puncture as shown in Figure 5.1. This not only gave significant error, but it did not allow us to compare event times, and the calculated energy of the final solution was significantly different from the other two methods. Further, without the rapid changes in geometry that accompany the event, the adaptive time stepping scheme did not

	Convex Splitting	Fully Implicit	Exponential Integrator
Error (l^2)	29.66	8.6836e-3	1.8772e-1
Maximum δt	1.8e-2	7.7e-4	1.0e-4
Computation Time (s)	194	43172	4540
Number of Time Steps	190	2997	21109
Event Time	none	0.9654	0.9658
Energy Drop	169.604	176.337	176.335

Table 5.1 Comparison of the convex splitting, fully implicit, and exponential integrator methods for a full simulation up to $T = 2$ with adaptive time stepping.

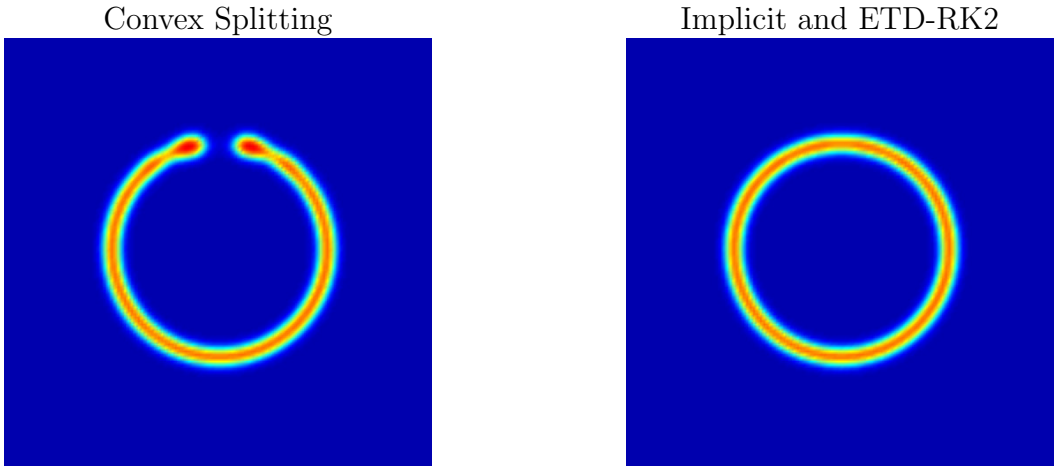


Figure 5.1 Final states of the complete simulation comparing the convex splitting method to the other two methods. The images show a two-dimensional slice of the three-dimensional solutions

slow down for that portion of the simulation and the computation time was absurdly fast.

The error from each simulation was calculated with the l^2-norm, and it shows that the implicit scheme was approximately twenty times more accurate than the ETD scheme. The implicit scheme also took time steps up to seven times larger than the ETD scheme. The big winner for the ETD method is that it was roughly ten times faster than the implicit scheme, and it accurately predicted the geometric event and ended with a computed energy very close to that of the implicit scheme.

The speed of the exponential time differencing was much faster than the other two methods. On average, the ETD scheme took 0.215 seconds per time step compared to 1.021

second and 14.405 seconds for the convex splitting and implicit schemes, respectively. This explains why the ETD scheme can successfully take far more small, accurate time steps. This speed difference is particularly important for the three-dimensional simulations. With the ETD scheme, we can compute important geometries out to $T = 300$ or less in one week. The same simulation with the convex splitting would likely miss the important dynamics, and the implicit scheme would require more than two months.

5.3 Fixed Time-Step Simulation

For a second comparison of the three methods, we conducted fixed time-step simulations for $\delta t = 4 \cdot 10^{-4}$, $2 \cdot 10^{-4}$, and $1 \cdot 10^{-4}$. For initial conditions, the simulation used the "true" solution from above at time $T = 1$, and we computed the solution up to $T = 2$. This initial geometry was shortly after the closing of the puncture and it evolved to resemble a hollow sphere.

The results are collected in Table 5.2, and they give further evidence that the convex splitting method has particularly bad evolution and thus poor accuracy. The implicit method is roughly two orders of magnitude more accurate than the ETD method for the time-step sizes used. The convex splitting scheme and the exponential integrator compute time steps at roughly the same speed, with the implicit scheme taking five to ten times longer. It is particularly interesting that when the step size halves (number of steps double), the amount of computation time does not double. Some of this can be attributed to the overhead of slow data transfer onto and off of the GPU card. However, the implicit scheme actually becomes more efficient for each time-step when the steps are small. This is because the initial guess for the Newton solve is much closer when the time-step is small, and it therefore uses fewer

	Convex Splitting		Fully Implicit		Exponential Integrator	
δt	Error	Time	Error	Time	Error	Time
4e-4	5.3619	603	1.2657e-3	6968	6.1249e-1	751
2e-4	4.1915	1016	6.1597e-4	8096	2.9415e-1	1126
1e-4	2.6570	1941	2.9146e-4	11973	1.0623e-1	2159

Table 5.2 Comparison of error and simulation time for fixed time-step simulations of the FCH equation. Error calculations use the l^2 norm, and time is given in seconds.

Newton iterations and far fewer preconditioned conjugate gradient iterations.

5.4 Time Step Size Restrictions

Although we have focused on time stepping methods that balance large time steps with accurate solutions and reasonable computation speeds, we have discovered that there are soft limits on the size of time steps for each of the three types of methods. Both the convex splitting method and the exponential time differencing give incorrect evolution for time steps that are too large. The implicit scheme is limited by the ability to complete the Newton convergence and the error control. In this section, we discuss the causes for these limitations and how to adjust for them.

5.4.1 Solution Freeze Out for Convex Splitting

In our work with Eyre's convex splitting method, we discovered that some numerical methods for the Cahn-Hilliard equation suffer from a behavior we call solution freeze out. This behavior occurs when doubling the time step size does not give twice the evolution of the solution, and the movement of mass with respect to time vanishes. This can happen when the higher frequency modes in the solution are over damped, causing the mass in the phases to have a limit on the distance evolved in a given time step.

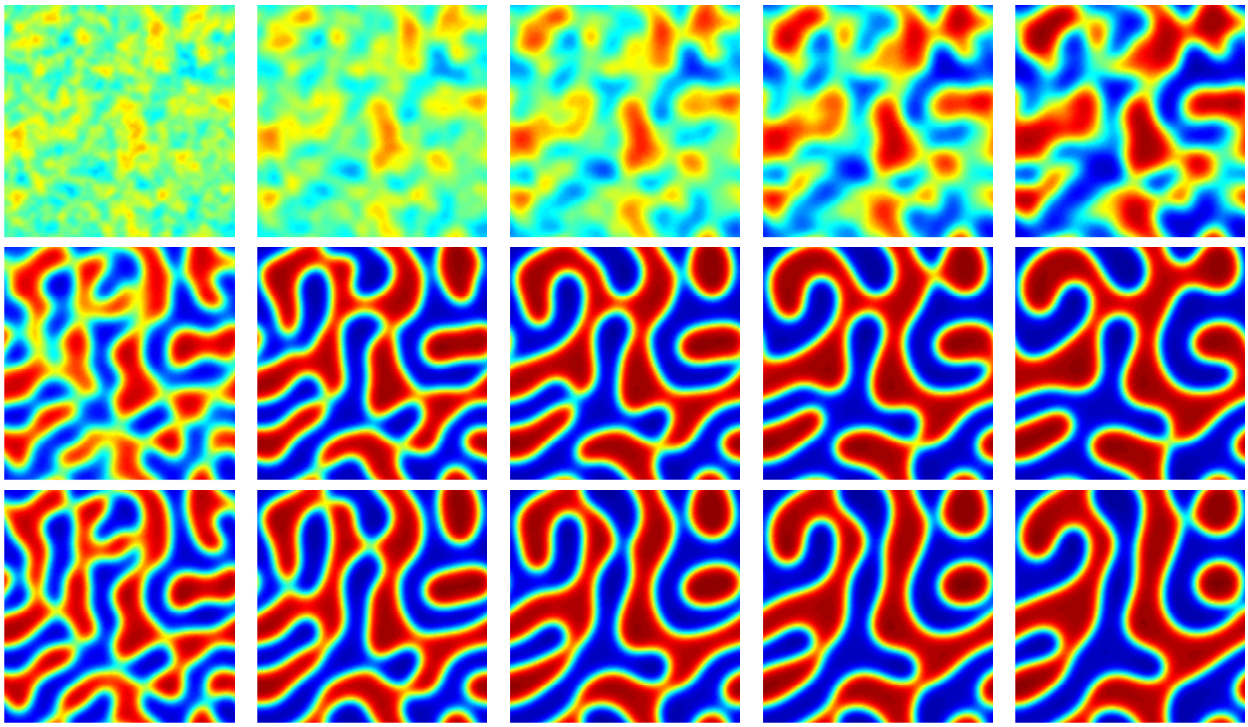


Figure 5.2 Time evolution showing freeze out of solution with larger time step sizes. For rows one through three $\delta t = 10^{-2}, 10^{-3},$ and 10^{-4} respectively.

We first noticed this behavior when performing a refinement study on an unstable equilibrium initial condition. When we increased the time step size by a factor of ten, the critical event happened nearly ten times later in the evolution. This was surprising because we expected any increase in error to drive the geometry away from the equilibrium point more quickly. Figure 5.2 shows this type of evolution for time step sizes $\delta t = 10^{-2}, 10^{-3}, \text{and} 10^{-4}$ in the coarsening stage of evolution for the CH equation. The coarsening of the solution should occur near $T = 2 \cdot 10^{-2}$, but is not captured for the largest time step size. Further, when $\delta t = 10^{-3}$, the freeze out leads to incorrect early topological changes which give a different solution at the final time. This freeze out behavior was also observed by Gomez and Hughes [55].

5.4.2 Solution Freeze Out for Exponential Time Differencing

The first order exponential integrator also experiences freeze out. This causes the decrease in accuracy leading to loss of convergence order which can be seen in Figures 4.1 and 4.2 as δt approaches 10^{-4}. The question becomes, at which time step size do we stop trusting the evolution of the solution? The obvious answer is to look at the order of convergence. Table 5.3 shows the time refinement study that gives the order of convergence for a given fixed δt used in the first order ETD scheme (4.3). For any $\delta t \gtrsim 10^{-4}$ the method ceases to be first order in time, and it will give incorrect evolution. The study was performed on a solution of Cahn-Hilliard equation that was far beyond the spinodal stage, but this behavior is consistent with long-time simulations of the FCH equation.

δt	Order
1.00E-3	-0.1754
5.00E-4	0.4640
2.50E-4	0.8845
1.25E-4	1.0263
6.25E-5	1.0433
3.12E-5	1.0354
1.56E-5	1.0266

Table 5.3 Order of convergence at given time step sizes.

More than just a refinement study tells us why $\delta t \approx 10^{-4}$ is the appropriate cutoff. If we look back at equation 4.3, we notice that all of the time evolution is embedded in the exponential operators. Freeze out happens when doubling the time step size results in less than twice the evolution. In Figure 5.3, we plot the amount of evolution versus time step size and wave number (as in 4.4). We define amount of evolution as the fractional change in

the solution divided by the fractional change in time. For the first order ETD method,

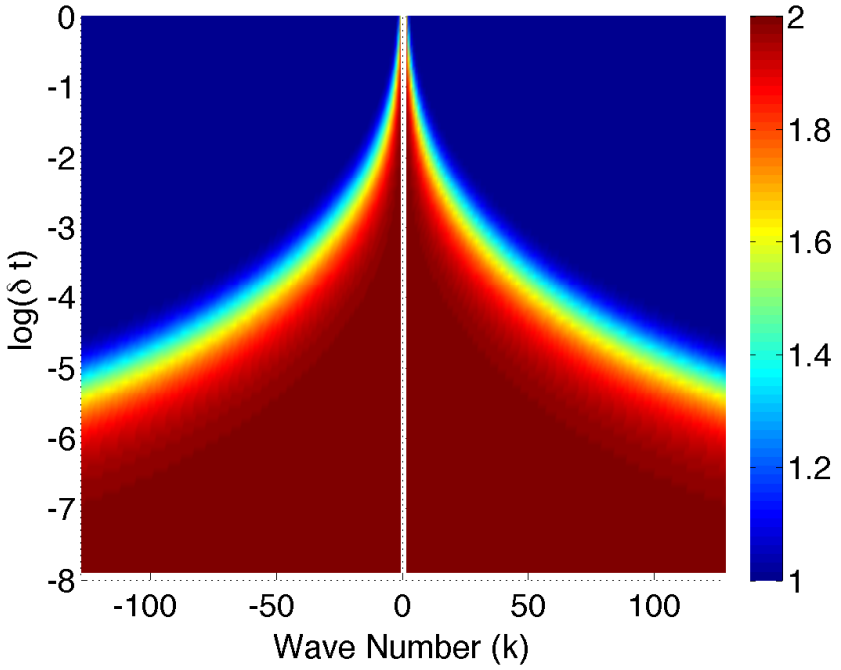
$$\text{amount of evolution} = \frac{\frac{\phi_1(2dt\mathcal{L})}{\phi_1(dt\mathcal{L})} - 1}{\frac{2dt}{dt} - 1} + 1.$$

As timestep size grows, note that the freeze out begins initially with high-frequency wave numbers. This behavior brings stability to the method, but when δt gets too large it starts damping out physically relevant wave numbers. We identify physically relevant wave numbers by the energy spectrum of the solution (Figure 5.4). For the geometry and length scales we compute, a typical simulation must capture $|k| \lesssim 40$, and from Figure 5.3, we see that this requires $\delta t \lesssim 10^{-4}$.

Switching now to the FCH equation, Figure 5.5 shows that the change of the linear operator increases the freeze out behavior, particularly for the convex splitting method. In two dimensions, the FCH equation requires accuracy for essentially the same wave numbers as the CH equation due to geometric similarities. Thus for the ETD schemes, we limit time steps to a maximum of $dt = 10^{-4}$ any time that the adaptive time stepping scheme estimates a time step larger than that. For the convex splitting scheme, the same limitation would require time steps no larger than $\delta t = 10^{-7}$ which greatly reduces its feasibility.

5.4.3 Implicit Time-step Size Restriction

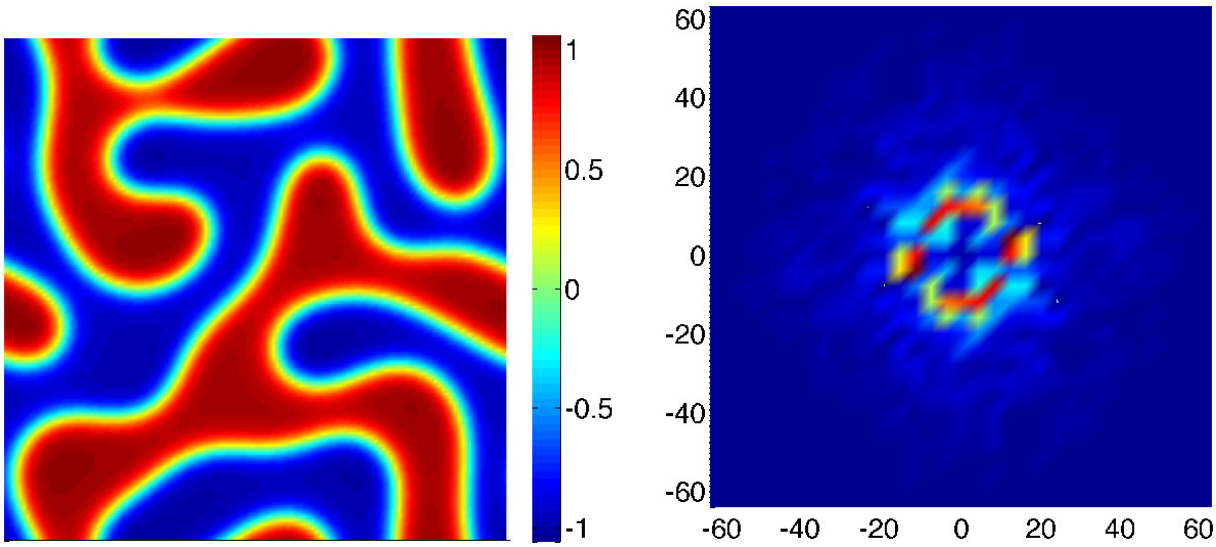
The implicit scheme has a more standard time-step size restriction. Due to the fourth-order nonlinearity in the Functionalized Cahn-Hilliard equation, Newton's method will not converge if the initial guess is not close enough. We have two reasonable choices for the initial guess; the solution at the last computed time step, or the solution of a forward Euler step. Preconditioning the conjugate gradient iteration has no effect on the time-step size



(a) Exponential Integrator

(b) Convex Splitting

Figure 5.3 Freeze out of operators for the two methods applied to the Cahn-Hilliard equation showing the amount of evolution when doubling the time step size. Note that both methods suffer from similar freeze out behavior.



(a) Typical Solution to the CH Equation (b) Energy Spectrum

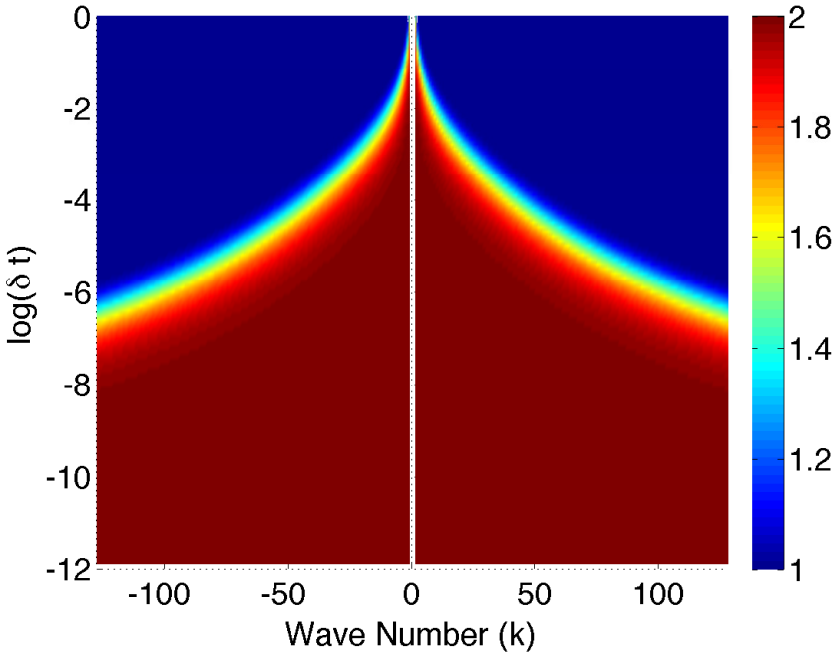
Figure 5.4 The energy spectrum of solutions to the Cahn-Hilliard equation is dominated by low wave numbers.

restriction.

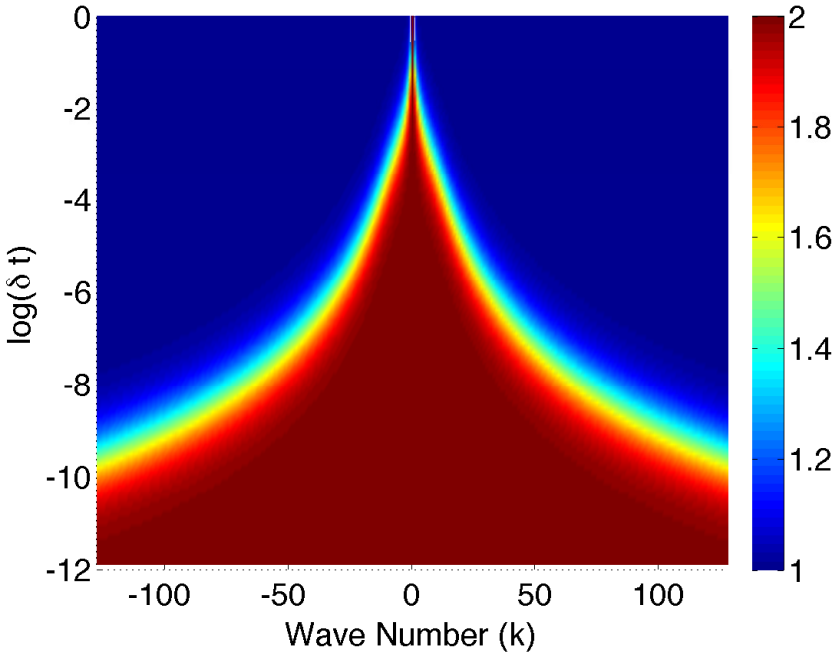
When using the last solution as the initial guess, the larger the step in time, the farther away the initial guess is from solution at the next time. This gives a time-step restriction on the order of 10^{-6} for the FCH equation with $\varepsilon = 0.03$. On the other hand, using a forward Euler step to initialize the implicit solve also has a limitation. The Euler step should capture the evolution and make the initial guess closer, but the explicit scheme is unstable for large time-steps and introduces spurious oscillations. This drives the forward Euler guess away from being a good initializer. It turns out that the forward Euler step is better in most cases and provides a time-step restriction on the order of 10^{-4} when $\varepsilon = 0.03$.

5.5 Conclusions

In conclusion, the exponential time differencing methods give the best combination of speed and accuracy, both of which are important in computing long time evolution for solutions



(a) Exponential Integrator



(b) Convex Splitting

Figure 5.5 Freeze out of operators for the two methods applied to the Functionalized Cahn-Hilliard equation showing the amount of evolution when doubling the time step size. Note that the convex splitting method suffers from much stronger freeze out behavior.

of the Functionalized Cahn-Hilliard equation in three dimensions. In combination with a spectral solution in space, application of the ETD method to a graphics processing unit greatly enhances the speed of the simulation and allows us to compute up to $T = 300$ in a week or less which is a great improvement over the fully implicit scheme. Although the ETD methods suffer from freeze out effects, the restriction is much less stringent than the convex splitting method. This gives time accuracy when the code enforces a hard limit on the size of the time-step. The ETD method captures relevant geometric evolution better than convex splitting and much faster than a fully implicit scheme.

The Functionalized Cahn-Hilliard equation can be effectively split using the convex splitting method proposed by Eyre [34, 46]. The scheme is very efficient computationally and performs its implicit iterations on a timescale much like an explicit method. A convex splitting scheme could be very effective in any calculation where the solution at final time is important, but the evolution that arrives at that solution is unneeded. The primary drawback is the lack of accuracy when time-steps are restrictively small. The convex splitting is also very difficult to adapt to new potential wells or vector phase problems which will be necessary when connecting with *ab initio* simulations and experimental work.

Lastly, the fully implicit scheme is the most accurate by at least an order of magnitude. If very precise evolution is required an implicit scheme is the proper choice. The cost of using the implicit scheme is its slow computation time. Preconditioning the Newton solve with a physics-based preconditioner does not alleviate the small time-step sizes required for the iterative solve to converge, but it does reduce the number of conjugate gradient iterations by up to a factor of one hundred and cuts the computation time similarly.

5.6 Future Work

As with any interesting research, there are a wide variety of directions we could take in the future. An interesting extension would be to improve the model to more than two phases. Some of the analysis has already been performed for three phases, and the new energy formulation comparable to equation 1.6 is

$$\mathcal{E}(u) = \int_\Omega \frac{1}{2} \left| \varepsilon^2 \Delta \vec{U} - W'\left(\vec{U}\right) + \varepsilon H_0 \begin{pmatrix} -u_2 \\ \\ u_1 \end{pmatrix} \right|^2 - \varepsilon \left[\eta_1 \frac{\varepsilon^2}{2} \left| \nabla \vec{U} \right|^2 + \eta_2 W\left(\vec{U}\right) \right] d\Omega. \quad (5.1)$$

$\vec{U} = (u_1, u_2)^T$ defines the fraction of polymer phases one and two, and the fraction of solvent is $u_s = 1 - u_1 - u_2$ so that the total amount is identically one. Further, the εH_0 term allows us to introduce intrinsic curvature and have greater control when modeling important biological systems such as endocytosis and exocytosis.

An important aspect of future work could be the development of better ties to the physics through a continuum mechanics description. Currently, there is evidence from experiments and some asymptotic analysis that suggests that the model is reasonable, but it would vastly improve the value of the FCH model if we could identify what types of physical systems connect well with the model's results. With this development, we would be able to identify the physical constants that each of the four coefficients in the FCH energy, namely ε, η_1, η_2, and τ. We would also need to obtain an accurate potential well, $W(u)$ for each specific system.

A more physics based application of the model is comparison with small angle scattering either with x-rays (SAXS) or neutrons (SANS). We have put some effort into this connection already, but the extension of the numerical to SAXS comparison could lead to an inverse

problem to find the four coefficients in the FCH energy. With some good fortune, this inverse problem could assist engineers and chemists in creating better membranes for PEM fuel cells, solid state Dye-Sensitized solar cells (DSSC), or polymer gel electrolyte batteries.

Finally, one of the widest areas of future research would be the generation of better numerical methods for handling different boundary conditions and larger domains. With the ability to compute solutions on larger domains with any type of boundary condition, we could simulate the entire pore network formation of Nafion or other materials from lab-like conditions. There would no longer be the restrictions of simulating inside bulk material with uniform water concentrations. The likely candidate for these changes is high order discontinuous Galerkin methods with global domain decomposition.

Although not exhaustive, this list gives a view of the opportunities for numerical computation that exist with the Functionalized Cahn-Hilliard model. The methods and implementations described deliver significant results, and valuable knowledge will come from the continuation of this research.

APPENDICES

Appendix A: Convex Splitting Code in CUDA

```
1  //
2  // spe6NL3Dcuda6.cu
3  //
4  // This code was written by Jaylan Jones to approximate solutions
       to the
5  // Functionalized Cahn-Hilliard Equation using the convex
       splitting
6  // scheme patterned after the work of Eyre.
7  //
8
9  #include <iostream>
10 #include <fstream>
11 #include <sstream>
12 #include <cuda.h>
13 #include <cufft.h>
14 #include <time.h>
15
16 #define PI 3.141592653589
17 #define REAL double
18 #define COMPLEX cufftDoubleComplex
19 #define TYPE CUFFT_Z2Z
20 #define EXEC cufftExecZ2Z
21
22 using namespace std;
23
24 //
25 // This series of kernels are all called in the main function, and
       they
26 // break up the computational work into pieces that have no data
27 // dependence on each other.
28 //
29
30 //
```

```
31 // build_shp constructs the approximate inverse to the Frechet
   derivative
32 // used in the iterative process that solves the implicit portion
   of
33 // the splitting.
34 //
35
36
37 __global__ void build_shp(REAL *shp, REAL *Lap, int n, REAL alpha,
    REAL eta2, REAL taub, REAL c1, REAL c2)
38 {
39   int idx = blockIdx.x*blockDim.x+threadIdx.x;
40   if(idx < n)
41     shp[idx] = -Lap[idx]*(alpha*alpha*Lap[idx]*(alpha*alpha*Lap[
    idx] - c2) + c1 + 1 - 0.5*taub*taub + alpha*eta2);
42 }
43
44 //
45 // All the numbered kernels are called in main
46 //
47
48 __global__ void kern0(int n, COMPLEX *U, COMPLEX *R, REAL evap,
    REAL dt)
49 {
50   int idx = blockIdx.x*blockDim.x+threadIdx.x;
51   if(idx < n)
52   {
53     U[idx].x = R[idx].x - dt * evap * (R[idx].x + 1) / 2;
54     U[idx].y = 0.0;
55   }
56 }
57
58 __global__ void kern1(int n, REAL alpha, REAL eta1, REAL eta2,
    REAL taub, REAL taus, COMPLEX *U, COMPLEX *F1U, COMPLEX *F2U,
    COMPLEX *F3R, COMPLEX *F4R, COMPLEX *F5R)
59 {
60   int idx = blockIdx.x*blockDim.x+threadIdx.x;
61   if(idx < n)
62   {
63     F1U[idx].x = 2.5*taub*pow(U[idx].x,4) + (0.5*taub*taub - 4 -
    alpha*eta2)*pow(U[idx].x,3);
64     F1U[idx].y = 0.0;
65
66     F2U[idx].x = (2 + alpha*eta1 - 0.5*taub*U[idx].x)*U[idx].x;
67     F2U[idx].y = 0.0;
```

```
68
69      F3R[idx].x = pow(U[idx].x,3);
70      F3R[idx].y = 0.0;
71
72      F4R[idx].x = (3*U[idx].x + taub)*U[idx].x;
73      F4R[idx].y = 0.0;
74
75      F5R[idx].x = 3*pow(U[idx].x,5) - 0.5*(taub*6 + alpha*eta2*taus
        )*pow(U[idx].x,2);
76      F5R[idx].y = 0.0;
77    }
78  }
79
80  __global__ void kern2(int n, COMPLEX *U, COMPLEX *LapR, REAL *Lap)
81  {
82    int idx = blockIdx.x*blockDim.x+threadIdx.x;
83    if(idx < n)
84    {
85      LapR[idx].x = Lap[idx]*U[idx].x;
86      LapR[idx].y = Lap[idx]*U[idx].y;
87    }
88  }
89
90  __global__ void kernDel(int n, COMPLEX *U, COMPLEX *DelxR, REAL *
        Delx, COMPLEX *DelyR, REAL *Dely, COMPLEX *DelzR, REAL *Delz)
91  {
92    int idx = blockIdx.x*blockDim.x+threadIdx.x;
93    if(idx < n)
94    {
95      DelxR[idx].x = -Delx[idx]*U[idx].y;
96      DelxR[idx].y = Delx[idx]*U[idx].x;
97
98      DelyR[idx].x = -Dely[idx]*U[idx].y;
99      DelyR[idx].y = Dely[idx]*U[idx].x;
100
101     DelzR[idx].x = -Delz[idx]*U[idx].y;
102     DelzR[idx].y = Delz[idx]*U[idx].x;
103   }
104 }
105
106 __global__ void kern3(int n, REAL *Lap, REAL alpha, COMPLEX *expR,
        COMPLEX *F1U, COMPLEX *F2U, COMPLEX *F4RLapR, COMPLEX *F4R,
        COMPLEX *LapR)
107 {
108   int idx = blockIdx.x*blockDim.x+threadIdx.x;
```

```
109   if(idx < n)
110   {
111     expR[idx].x = -Lap[idx]*(alpha*alpha*F2U[idx].x*Lap[idx] + F1U
      [idx].x);
112     expR[idx].y = -Lap[idx]*(alpha*alpha*F2U[idx].y*Lap[idx] + F1U
      [idx].y);
113
114     F4RLapR[idx].x = F4R[idx].x*LapR[idx].x/n;
115     F4RLapR[idx].y = 0.0;
116   }
117 }
118
119 //
120 // kernE computes the FCH energy from the solution at the current
      time
121 //
122
123 __global__ void kernE(int n, REAL alpha, REAL eta1, REAL eta2,
      REAL taub, REAL taus, COMPLEX *LapR, COMPLEX *R, COMPLEX *DelxR
      , COMPLEX *DelyR, COMPLEX *DelzR, REAL *Energy)
124 {
125   int idx = blockIdx.x*blockDim.x+threadIdx.x;
126   if(idx < n)
127   {
128     Energy[idx] = 0.5*(alpha*alpha*LapR[idx].x/n - (R[idx].x*R[idx
      ].x-1)*(R[idx].x+taub/2))*(alpha*alpha*LapR[idx].x/n - (R[idx].
      x*R[idx].x-1)*(R[idx].x+taub/2)) - alpha*0.5*(alpha*alpha*eta1
      *(DelxR[idx].x/n*DelxR[idx].x/n + DelyR[idx].x/n*DelyR[idx].x/n
       + DelzR[idx].x/n*DelzR[idx].x/n) + eta2*(R[idx].x+1)*(R[idx].x
      +1)*(0.5*(R[idx].x-1)*(R[idx].x-1)+taus/3*(R[idx].x-2)));
129   }
130 }
131
132 __global__ void kern4(int n, COMPLEX *F4RLapR, COMPLEX *F4R,
      COMPLEX *LapR)
133 {
134   int idx = blockIdx.x*blockDim.x+threadIdx.x;
135   if(idx < n)
136   {
137     F4RLapR[idx].x = F4R[idx].x*LapR[idx].x/n;
138     F4RLapR[idx].y = 0.0;
139
140   }
141 }
142
```

```
143 __global__ void kern5(int n, REAL *Lap, REAL alpha, REAL eta2,
    REAL taub, COMPLEX *hr, COMPLEX *expR, COMPLEX *F3R, COMPLEX *
    F4RLapR, COMPLEX *U, COMPLEX *F5R)
144 {
145   int idx = blockIdx.x*blockDim.x+threadIdx.x;
146   if(idx < n)
147   {
148     hr[idx].x = expR[idx].x - Lap[idx]*(alpha*alpha*Lap[idx]*(
    alpha*alpha*Lap[idx]*U[idx].x-F3R[idx].x) - alpha*alpha*F4RLapR
    [idx].x + (1-0.5*taub*taub+alpha*eta2)*U[idx].x + F5R[idx].x);
149     hr[idx].y = expR[idx].y - Lap[idx]*(alpha*alpha*Lap[idx]*(
    alpha*alpha*Lap[idx]*U[idx].y-F3R[idx].y) - alpha*alpha*F4RLapR
    [idx].y + (1-0.5*taub*taub+alpha*eta2)*U[idx].y + F5R[idx].y);
150   }
151 }
152
153 //
154 // warpReduce, reduce, warpReduceSum, and reduceSum parallelize
    the reductions necessary to calculate energy and change in
    solution
155 //
156
157 __device__ void warpReduce(volatile REAL *sdata, unsigned int tid,
     int blockSize)
158 {
159   if (blockSize >= 64) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 32]));
160   if (blockSize >= 32) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 16]));
161   if (blockSize >= 16) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 8]));
162   if (blockSize >= 8) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 4]));
163   if (blockSize >= 4) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 2]));
164   if (blockSize >= 2) sdata[tid] = max(fabs(sdata[tid]), fabs(
    sdata[tid + 1]));
165 }
166
167 __global__ void reduce(COMPLEX *g_idata, COMPLEX *g_odata, int n,
    int blockSize)
168 {
169   extern __shared__ REAL sdata[];
170   unsigned int tid = threadIdx.x;
171   unsigned int i = blockIdx.x*(blockSize*2) + tid;
```

```
172   unsigned int gridSize = blockSize*2*gridDim.x;
173   sdata[tid] = 0;
174
175   while (i < n) {sdata[tid] = max(fabs(g_idata[i].x),fabs(g_idata[
       i+blockSize].x)); i += gridSize; }
176   __syncthreads();
177
178   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 256])); } __syncthreads(); }
179   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 128])); } __syncthreads(); }
180   if (blockSize >= 128) { if (tid < 64) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 64])); } __syncthreads(); }
181   if (tid < 32) warpReduce(sdata, tid, blockSize);
182   if (tid == 0) g_odata[blockIdx.x].x = sdata[0];
183 }
184
185 __device__ void warpReduceSum(volatile REAL *sdata, unsigned int
       tid, int blockSize)
186 {
187   if (blockSize >= 64) sdata[tid] = sdata[tid] + sdata[tid + 32];
188   if (blockSize >= 32) sdata[tid] = sdata[tid] + sdata[tid + 16];
189   if (blockSize >= 16) sdata[tid] = sdata[tid] + sdata[tid + 8];
190   if (blockSize >= 8) sdata[tid] = sdata[tid] + sdata[tid + 4];
191   if (blockSize >= 4) sdata[tid] = sdata[tid] + sdata[tid + 2];
192   if (blockSize >= 2) sdata[tid] = sdata[tid] + sdata[tid + 1];
193 }
194
195 __global__ void reduceSum(REAL *g_idata, REAL *g_odata, int n, int
       blockSize)
196 {
197   extern __shared__ REAL sdata[];
198   unsigned int tid = threadIdx.x;
199   unsigned int i = blockIdx.x*(blockSize*2) + tid;
200   unsigned int gridSize = blockSize*2*gridDim.x;
201   sdata[tid] = 0;
202
203   while (i < n) {sdata[tid] = g_idata[i] + g_idata[i+blockSize]; i
       += gridSize; }
204   __syncthreads();
205
206   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = sdata[tid]
       + sdata[tid + 256]; } __syncthreads(); }
207   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = sdata[tid]
       + sdata[tid + 128]; } __syncthreads(); }
```

```
208   if (blockSize >= 128) { if (tid < 64) { sdata[tid] = sdata[tid]
      + sdata[tid + 64]; } __syncthreads(); }
209   if (tid < 32) warpReduceSum(sdata, tid, blockSize);
210   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
211 }
212
213 __global__ void kern6(int n, REAL dt, COMPLEX *hp, REAL *shp,
      COMPLEX *expR, COMPLEX *U)
214 {
215   int idx = blockIdx.x*blockDim.x+threadIdx.x;
216   if(idx < n)
217   {
218       hp[idx].x = 1/dt + shp[idx];
219
220       expR[idx].x = -U[idx].x/dt + expR[idx].x;
221       expR[idx].y = -U[idx].y/dt + expR[idx].y;
222   }
223 }
224
225 __global__ void kern7(int n, REAL alpha, REAL eta2, REAL taub,
      REAL taus, COMPLEX *R, COMPLEX *F3R, COMPLEX *F4R, COMPLEX *F5R
      )
226 {
227   int idx = blockIdx.x*blockDim.x+threadIdx.x;
228   if(idx < n)
229   {
230     F3R[idx].x = pow(R[idx].x,3);
231     F3R[idx].y = 0.0;
232
233     F4R[idx].x = (3*R[idx].x + taub)*R[idx].x;
234     F4R[idx].y = 0.0;
235
236     F5R[idx].x = 3*pow(R[idx].x,5) - 0.5*(taub*6 + alpha*eta2*taus
      )*pow(R[idx].x,2);
237     F5R[idx].y = 0.0;
238   }
239 }
240
241 __global__ void kern8(int n, REAL dt, REAL alpha, REAL eta2, REAL
      taub, REAL *Lap, COMPLEX *change, COMPLEX *expR, COMPLEX *R,
      COMPLEX *F3R, COMPLEX *F4RLapR, COMPLEX *F5R, COMPLEX *hp)
242 {
243   int idx = blockIdx.x*blockDim.x+threadIdx.x;
244   if(idx < n)
245   {
```

```
246       change[idx].x = expR[idx].x + R[idx].x/dt - Lap[idx]*(alpha*
      alpha*Lap[idx]*(alpha*alpha*Lap[idx]*R[idx].x-F3R[idx].x) -
      alpha*alpha*F4RLapR[idx].x + (1-0.5*taub*taub+alpha*eta2)*R[idx
      ].x + F5R[idx].x);
247       change[idx].y = expR[idx].y + R[idx].y/dt - Lap[idx]*(alpha*
      alpha*Lap[idx]*(alpha*alpha*Lap[idx]*R[idx].y-F3R[idx].y) -
      alpha*alpha*F4RLapR[idx].y + (1-0.5*taub*taub+alpha*eta2)*R[idx
      ].y + F5R[idx].y);
248
249       change[idx].x = -change[idx].x/hp[idx].x;
250       change[idx].y = -change[idx].y/hp[idx].x;
251     }
252 }
253
254 __global__ void kern9(int n, COMPLEX *R, COMPLEX *change)
255 {
256   int idx = blockIdx.x*blockDim.x+threadIdx.x;
257   if(idx < n)
258   {
259     R[idx].x = (R[idx].x + change[idx].x)/n;
260     R[idx].y = 0.0;
261   }
262 }
263
264 //
265 // The node structure builds a linked list that collects energy
      and time data
266 //
267
268 struct node
269 {
270   REAL time;
271   REAL energy;
272   struct node* next;
273 };
274
275 void addNode(struct node*& tail, REAL time)
276 {
277   struct node* newNode = new node;
278   newNode->time = time;
279   newNode->energy = 0.0;
280   newNode->next = NULL;
281   tail->next = newNode;
282   tail = newNode;
283 }
```

```
284
285 //
286 // This is where the main function begins. To make sense of the
      code, begin here.
287 //
288
289 int main (int argc, char * const argv[]) {
290   if (argc <=1 )
291   {
292     cout << "Usage: " << argv[0] << " <filename> <output off/
      increasing/even (0/1/2)>" << endl;
293     exit(1);
294   }
295
296   clock_t tic, toc;
297   tic = clock();
298   //
299   // Initialize necessary constants
300   //
301   cout.precision(10);
302   int i, j, k;
303   fstream myfile, myfile2, myfile3, myfile4;
304   stringstream sstm;
305   char datain[50];
306
307   REAL alpha, eta1, eta2, time, tmax, dt, tol, Lx, Ly, Lz, c1, c2,
        ct, taub, taus, evap, tpicstep;
308   int m, n, Nx, Ny, Nz, Mx, My, Mz, blocksize;
309   COMPLEX *max, *max_h, *res;
310
311   max_h = new COMPLEX;
312   res = new COMPLEX;
313
314   struct node* tEhead = NULL;
315   struct node* tEtail = NULL;
316
317   tEhead = new node;
318   tEhead->time = 0.0;
319   tEhead->energy = 0.0;
320   tEtail = tEhead;
321
322   string filein, fileUout, filetEout;
323
324   //
```

```
325   // Read in all the necessary data from the file given by the
       user
326   //
327
328   myfile3.open(argv[1], ios::in);
329
330   if(myfile3.is_open())
331   {
332     myfile3.ignore(512,'=');
333     myfile3 >> datain;
334     filein = datain;
335     cout << "Init cond file = " << filein << endl;
336
337     myfile3.ignore(512,'=');
338     myfile3 >> datain;
339     fileUout = datain;
340     cout << "Write out solution file = " << fileUout << ".dat" <<
       endl;
341
342     myfile3.ignore(512,'=');
343     myfile3 >> datain;
344     filetEout = datain;
345     cout << "Write out time and energy file = " << filetEout << ".
       dat" << endl;
346
347     myfile3.ignore(512,'=');
348     myfile3 >> datain;
349     alpha = atof(datain);
350     cout << "alpha = " << alpha << endl;
351
352     myfile3.ignore(512,'=');
353     myfile3 >> datain;
354     taub = atof(datain);
355     cout << "taub = " << taub << endl;
356
357     myfile3.ignore(512,'=');
358     myfile3 >> datain;
359     taus = atof(datain);
360     cout << "taus = " << taus << endl;
361
362     myfile3.ignore(512,'=');
363     myfile3 >> datain;
364     eta1 = atof(datain);
365     cout << "eta1 = " << eta1 << endl;
366
```

```
367     myfile3.ignore(512,'=');
368     myfile3 >> datain;
369     eta2 = atof(datain);
370     cout << "eta2 = " << eta2 << endl;
371
372     myfile3.ignore(512,'=');
373     myfile3 >> datain;
374     evap = atof(datain);
375     cout << "evap = " << evap << endl;
376
377     myfile3.ignore(512,'=');
378     myfile3 >> datain;
379     tmax = atof(datain);
380     cout << "tmax = " << tmax << endl;
381
382     myfile3.ignore(512,'=');
383     myfile3 >> datain;
384     dt = atof(datain);
385     cout << "dt = " << dt << endl;
386
387     myfile3.ignore(512,'=');
388     myfile3 >> datain;
389     tpicstep = atof(datain);
390     cout << "tpicstep = " << tpicstep << endl;
391
392     myfile3.ignore(512,'=');
393     myfile3 >> datain;
394     Nx = atoi(datain);
395     cout << "Nx = " << Nx << endl;
396
397     myfile3.ignore(512,'=');
398     myfile3 >> datain;
399     Ny = atoi(datain);
400     cout << "Ny = " << Ny << endl;
401
402     myfile3.ignore(512,'=');
403     myfile3 >> datain;
404     Nz = atoi(datain);
405     cout << "Nz = " << Nz << endl;
406
407     myfile3.ignore(512,'=');
408     myfile3 >> datain;
409     Lx = atof(datain);
410     cout << "Lx = " << Lx << endl;
411
```

```
412     myfile3.ignore(512,'=');
413     myfile3 >> datain;
414     Ly = atof(datain);
415     cout << "Ly = " << Ly << endl;
416
417     myfile3.ignore(512,'=');
418     myfile3 >> datain;
419     Lz = atof(datain);
420     cout << "Lz = " << Lz << endl;
421
422     myfile3.ignore(512,'=');
423     myfile3 >> datain;
424     c1 = atof(datain);
425     cout << "c1 = " << c1 << endl;
426
427     myfile3.ignore(512,'=');
428     myfile3 >> datain;
429     c2 = atof(datain);
430     cout << "c2 = " << c2 << endl;
431
432     myfile3.ignore(512,'=');
433     myfile3 >> datain;
434     ct = atof(datain);
435     cout << "ct = " << ct << endl;
436
437     myfile3.ignore(512,'=');
438     myfile3 >> datain;
439     tol = atof(datain);
440     cout << "tol = " << tol << endl;
441
442     myfile3.ignore(512,'=');
443     myfile3 >> datain;
444     blocksize = atoi(datain);
445     cout << "Block size = " << blocksize << endl;
446     cout << "Grid size = " << Nx*Ny*Nz/blocksize << endl;
447   }
448   else { cout << "File could not be read." << endl; }
449
450   time = 0.0;
451   REAL tPic = 1.0;
452
453   if (*argv[2] == '2') {tPic = tpicstep;}
454
455   Mx = Nx/2;
456   My = Ny/2;
```

```
457   Mz = Nz/2;
458   n = Nx*Ny*Nz;
459   dim3 dimBlock(blocksize);
460   dim3 dimGrid(n/dimBlock.x);
461
462   //
463   // Build the laplacian array, and dynamically allocate other
       needed arrays
464   //
465
466   REAL *xkvec, *ykvec, *zkvec;
467   xkvec = new REAL[Nx];
468   ykvec = new REAL[Ny];
469   zkvec = new REAL[Nz];
470
471   for (i=0; i<Mx; i++)
472   {
473     xkvec[i] = i*PI/Lx;
474     xkvec[i+Mx] = (i-Mx)*PI/Lx;
475   }
476
477   for (i=0; i<My; i++)
478   {
479     ykvec[i] = i*PI/Ly;
480     ykvec[i+My] = (i-My)*PI/Ly;
481   }
482
483   for (i=0; i<Mz; i++)
484   {
485     zkvec[i] = i*PI/Lz;
486     zkvec[i+Mz] = (i-Mz)*PI/Lz;
487   }
488
489   REAL *Lap_h, *Lap, *shp, *Energy, *Energy_h;
490   REAL *Delx_h, *Dely_h, *Delz_h, *Delx, *Dely, *Delz;
491   Energy_h = new REAL;
492   Lap_h = new REAL[n];
493   Delx_h = new REAL[n];
494   Dely_h = new REAL[n];
495   Delz_h = new REAL[n];
496
497   for (i=0; i<Nx; i++) {
498     for (j=0; j<Ny; j++) {
499       for (k=0; k<Nz; k++)
500       {
```

```
501           Lap_h[k+Nz*(j+Ny*i)] = -(xkvec[i]*xkvec[i]+ykvec[j]*ykvec[
    j]+zkvec[k]*zkvec[k]);
502           Delx_h[k+Nz*(j+Ny*i)] = xkvec[i];
503           Dely_h[k+Nz*(j+Ny*i)] = ykvec[j];
504           Delz_h[k+Nz*(j+Ny*i)] = zkvec[k];
505         }
506       }
507     }
508
509     cout << "Lap built" << endl;
510
511     int nBytes = sizeof(COMPLEX)*n;
512     cout << "nBytes is = " << nBytes << endl;
513     COMPLEX *U_h, *U, *R, *F1U, *F2U, *F3R, *F4R, *F5R, *LapR, *
    DelxR, *DelyR, *DelzR, *expR, *F4RLapR, *hr, *hp,  *change;
514     U_h = new COMPLEX[nBytes];
515
516     //
517     // Read in initial U data from filein
518     //
519
520     myfile.open(filein.c_str(),ios::in);
521
522     if(myfile.is_open())
523     {
524       for (i=0; i<n; i++)
525       {
526         myfile.getline(datain, 50);
527         U_h[i].x = atof(datain);
528         U_h[i].y = 0.0;
529       }
530       cout << "Initial data read in successfully" << endl;
531     } else { cout << "Initial data not read" << endl; }
532
533     myfile.close();
534
535     //
536     // Allocate memory on the GPU and set up space for the Fourier
    transforms
537     //
538
539     cudaMalloc((void **)&Lap, sizeof(REAL)*n);
540     cudaMalloc((void **)&Delx, sizeof(REAL)*n);
541     cudaMalloc((void **)&Dely, sizeof(REAL)*n);
542     cudaMalloc((void **)&Delz, sizeof(REAL)*n);
```

```
543   cudaMalloc((void **)&max, sizeof(COMPLEX)*dimGrid.x/2);
544
545   cudaMalloc((void **)&shp, sizeof(REAL)*n);
546   cudaMalloc((void **)&Energy, sizeof(REAL)*n);
547   cudaMalloc((void **)&U, nBytes);
548   cudaMalloc((void **)&R, nBytes);
549   cudaMalloc((void **)&F1U, nBytes);
550   cudaMalloc((void **)&F2U, nBytes);
551   cudaMalloc((void **)&F3R, nBytes);
552   cudaMalloc((void **)&F4R, nBytes);
553   cudaMalloc((void **)&F5R, nBytes);
554   cudaMalloc((void **)&LapR, nBytes);
555   cudaMalloc((void **)&DelxR, nBytes);
556   cudaMalloc((void **)&DelyR, nBytes);
557   cudaMalloc((void **)&DelzR, nBytes);
558   cudaMalloc((void **)&expR, nBytes);
559   cudaMalloc((void **)&F4RLapR, nBytes);
560   cudaMalloc((void **)&hr, nBytes);
561   cudaMalloc((void **)&hp, nBytes);
562   cudaMalloc((void **)&change, nBytes);
563   cout << "Device memory allocated" << endl;
564
565   cudaMemcpy(Lap, Lap_h, sizeof(REAL)*n, cudaMemcpyHostToDevice);
566   cudaMemcpy(Delx, Delx_h, sizeof(REAL)*n, cudaMemcpyHostToDevice)
       ;
567   cudaMemcpy(Dely, Dely_h, sizeof(REAL)*n, cudaMemcpyHostToDevice)
       ;
568   cudaMemcpy(Delz, Delz_h, sizeof(REAL)*n, cudaMemcpyHostToDevice)
       ;
569   cudaMemcpy(R, U_h, nBytes, cudaMemcpyHostToDevice);
570   cout << "Initial data copied" << endl;
571
572   cufftHandle plan;
573   cufftPlan3d(&plan, Nx, Ny, Nz, TYPE);
574
575   build_shp<<< dimGrid, dimBlock>>>(shp, Lap, n, alpha, eta2, taub
       , c1, c2);
576
577   kern0<<< dimGrid, dimBlock>>>(n, U, R, 0, 0);
578
579   sstm << filetEout << ".dat";
580   myfile4.open(sstm.str().c_str(),ios::out);
581   myfile4.precision(14);
582   sstm.str("");
583
```

```
584   //
585   // while loop governing time stepping
586   //
587   while(time < tmax)
588   {
589     //
590     // Compute the explicit portion of the right hand side
591     //
592     res[0].x = 1000.0;
593
594     kern1<<< dimGrid, dimBlock>>>(n, alpha, eta1, eta2, taub, taus
      , R, F1U, F2U, F3R, F4R, F5R);
595
596     EXEC(plan, U, U, CUFFT_FORWARD);
597
598     kern2<<< dimGrid, dimBlock>>>(n, U, LapR, Lap);
599
600     kernDel<<< dimGrid, dimBlock>>>(n, U, DelxR, Delx, DelyR, Dely
      , DelzR, Delz);
601
602     EXEC(plan, F1U, F1U, CUFFT_FORWARD);
603     EXEC(plan, F2U, F2U, CUFFT_FORWARD);
604     EXEC(plan, LapR, LapR, CUFFT_INVERSE);
605     EXEC(plan, DelxR, DelxR, CUFFT_INVERSE);
606     EXEC(plan, DelyR, DelyR, CUFFT_INVERSE);
607     EXEC(plan, DelzR, DelzR, CUFFT_INVERSE);
608
609     kern3<<< dimGrid, dimBlock>>>(n, Lap, alpha, expR, F1U, F2U,
    F4RLapR, F4R, LapR);
610
611     //
612     // Compute the energy of the solution from the previous time
    step
613     //
614
615     kernE<<< dimGrid, dimBlock>>>(n, alpha, eta1, eta2, taub, taus
      , LapR, R, DelxR, DelyR, DelzR, Energy);
616     cudaMemcpy(Energy_h, Energy, sizeof(REAL),
    cudaMemcpyDeviceToHost);
617
618     reduceSum<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(
    Energy, Energy, n, blocksize);
619     cudaMemcpy(Energy_h, Energy, sizeof(REAL),
    cudaMemcpyDeviceToHost);
620     m = n/(blocksize*2);
```

```
621     while (m > 1)
622     {
623       reduceSum<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(
    Energy, Energy, m, blocksize);
624       m = m/(blocksize*2);
625     }
626
627     cudaMemcpy(Energy_h, Energy, sizeof(REAL),
    cudaMemcpyDeviceToHost);
628     tEtail->energy = *Energy_h;
629     myfile4 << tEtail->time << "\t" << tEtail->energy << endl;
630
631     //
632     // Compute the difference between the explicit step and the
    previous time step to approximate the adaptive time stepping.
633     //
634     EXEC(plan, F3R, F3R, CUFFT_FORWARD);
635     EXEC(plan, F4RLapR, F4RLapR, CUFFT_FORWARD);
636     EXEC(plan, F5R, F5R, CUFFT_FORWARD);
637
638     kern5<<< dimGrid, dimBlock>>>(n, Lap, alpha, eta2, taub, hr,
    expR, F3R, F4RLapR, U, F5R);
639
640     EXEC(plan, hr, hr, CUFFT_INVERSE);
641
642     reduce<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(hr, max
    , n, blocksize);
643     m = n/(blocksize*2);
644     while (m > 1)
645     {
646       reduce<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(max,
    max, m, blocksize);
647       m = m/(blocksize*2);
648     }
649
650     cudaMemcpy(max_h, max, sizeof(REAL), cudaMemcpyDeviceToHost);
651     max_h[0].x = max_h[0].x / n;
652     dt = ct/max_h[0].x;
653
654     //
655     // Adjust the timestep if necessary
656     //
657     if (dt < 0.000001)
658     {
659       dt = 0.000001;
```

```
660     }
661
662     if (dt > (tmax - time))
663     {
664       dt = tmax - time;
665     }
666
667     if (dt > (tPic - time) && *argv[2] != '0')
668     {
669       dt = tPic - time;
670     }
671
672     kern6<<< dimGrid, dimBlock>>>(n, dt, hp, shp, expR, U);
673
674     //
675     // This loop iterates until the nonlinear part is within
      tolerance of the solution
676     //
677     while (res[0].x > tol)
678     {
679       kern7<<< dimGrid, dimBlock>>>(n, alpha, eta2, taub, taus, R,
       F3R, F4R, F5R);
680
681       EXEC(plan, R, R, CUFFT_FORWARD);
682
683       kern2<<< dimGrid, dimBlock>>>(n, R, LapR, Lap);
684
685       EXEC(plan, LapR, LapR, CUFFT_INVERSE);
686
687       kern4<<< dimGrid, dimBlock>>>(n, F4RLapR, F4R, LapR);
688
689       EXEC(plan, F3R, F3R, CUFFT_FORWARD);
690       EXEC(plan, F4RLapR, F4RLapR, CUFFT_FORWARD);
691       EXEC(plan, F5R, F5R, CUFFT_FORWARD);
692
693       kern8<<< dimGrid, dimBlock>>>(n, dt, alpha, eta2, taub, Lap,
       change, expR, R, F3R, F4RLapR, F5R, hp);
694
695       EXEC(plan, change, change, CUFFT_INVERSE);
696
697       reduce<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(
      change, max, n, blocksize);
698       m = n/(blocksize*2);
699       while (m > 1)
700       {
```

```
701         reduce<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(max
    , max, m, blocksize);
702         m = m/(blocksize*2);
703       }
704
705       cudaMemcpy(res, max, sizeof(REAL), cudaMemcpyDeviceToHost);
706       res[0].x = res[0].x / (REAL) n;
707
708       EXEC(plan, R, R, CUFFT_INVERSE);
709
710       kern9<<< dimGrid, dimBlock>>>(n, R, change);
711     }
712
713     //
714     // Update time and update U to be the solution at the next
    time step
715     // Write out solution to hard disk if required
716     //
717     kern0<<< dimGrid, dimBlock>>>(n, U, R, evap, dt);
718
719     time = time + dt;
720
721     if (time == tPic && *argv[2] != '0')
722     {
723       cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
724       sstm << fileUout << (int) tPic << ".dat";
725       myfile2.open(sstm.str().c_str(),ios::out);
726       myfile2.precision(14);
727       if (myfile2.is_open())
728       {
729
730         for (i=0; i<n; i++)
731         {
732           myfile2 << U_h[i].x << endl;
733         }
734
735       }
736       cout << "Writing out " << sstm.str() << endl;
737       myfile2.close();
738       sstm.str("");
739
740       if (*argv[2] == '1')
741       {
742         tPic += pow((REAL) 10.0,(REAL) ceil(log10(tPic+1))-1);
743       }
```

```
744         else
745         {
746           tPic += tpicstep;
747         }
748       }
749
750       cout << time << "\t" << dt << "\t" << tEtail->energy << endl;
751       addNode(tEtail, time);
752     }
753
754     //
755     // Write final solution to hard disk and compute its energy
756     //
757     cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
758
759     EXEC(plan, R, R, CUFFT_FORWARD);
760
761     kern2<<< dimGrid, dimBlock>>>(n, R, LapR, Lap);
762
763     kernDel<<< dimGrid, dimBlock>>>(n, R, DelxR, Delx, DelyR, Dely,
       DelzR, Delz);
764
765     EXEC(plan, LapR, LapR, CUFFT_INVERSE);
766     EXEC(plan, DelxR, DelxR, CUFFT_INVERSE);
767     EXEC(plan, DelyR, DelyR, CUFFT_INVERSE);
768     EXEC(plan, DelzR, DelzR, CUFFT_INVERSE);
769
770     kernE<<< dimGrid, dimBlock>>>(n, alpha, eta1, eta2, taub, taus,
       LapR, U, DelxR, DelyR, DelzR, Energy);
771
772     reduceSum<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(Energy
       , Energy, n, blocksize);
773     m = n/(blocksize*2);
774     while (m > 1)
775     {
776       reduceSum<<< dimGrid, dimBlock, blocksize*sizeof(REAL)>>>(
       Energy, Energy, m, blocksize);
777       m = m/(blocksize*2);
778     }
779
780     cudaMemcpy(Energy_h, Energy, sizeof(REAL),
       cudaMemcpyDeviceToHost);
781     tEtail->energy = *Energy_h;
782
783     myfile4 << tEtail->time << "\t" << tEtail->energy << endl;
```

```
784   myfile4.close();
785
786   sstm << fileUout << (int) tmax << ".dat";
787   myfile2.open(sstm.str().c_str(),ios::out);
788   myfile2.precision(14);
789   if (myfile2.is_open())
790   {
791
792     for (i=0; i<n; i++)
793     {
794       myfile2 << U_h[i].x << endl;
795     }
796
797   }
798   cout << "Writing out " << sstm.str() << endl;
799   myfile2.close();
800   sstm.str("");
801
802   while (tEhead != NULL)
803   {
804     tEtail = tEhead;
805     tEhead = tEhead->next;
806     delete tEtail;
807   }
808
809   //
810   // Perform clean up of memory used
811   //
812
813   free(U_h);
814   free(max_h);
815   free(Energy_h);
816   free(res);
817
818   cudaFree(Lap);
819   cudaFree(Delx);
820   cudaFree(Dely);
821   cudaFree(Delz);
822   cudaFree(max);
823   cudaFree(Energy);
824   cudaFree(shp);
825   cudaFree(U);
826   cudaFree(R);
827   cudaFree(F1U);
828   cudaFree(F2U);
```

```
829   cudaFree(F3R);
830   cudaFree(F4R);
831   cudaFree(F5R);
832   cudaFree(LapR);
833   cudaFree(DelxR);
834   cudaFree(DelyR);
835   cudaFree(DelzR);
836   cudaFree(expR);
837   cudaFree(F4RLapR);
838   cudaFree(hr);
839   cudaFree(hp);
840   cudaFree(change);
841
842   delete[] xkvec;
843   delete[] ykvec;
844   delete[] zkvec;
845   delete[] Lap_h;
846   delete[] Delx_h;
847   delete[] Dely_h;
848   delete[] Delz_h;
849
850   toc = clock() - tic;
851   cout << "\n" << toc/ ((double) CLOCKS_PER_SEC) << endl;
852
853     return 0;
854 }
```

./code/spe6NL3Dcuda6.cu

Appendix B: Fully Implicit Code in CUDA

```
1  //
2  // FCHImp.cu
3  //
4  // This code was written by Jaylan Jones to approximate solutions
       to the
5  // Functionalized Cahn-Hilliard Equation using the fully implicit
       scheme.
6  //
7
8  #include <iostream>
9  #include <fstream>
10 #include <sstream>
11 #include <cuda.h>
12 #include <cufft.h>
13 #include <time.h>
14
15 #define PI 3.141592653589
16 #define REAL double
17 #define COMPLEX cufftDoubleComplex
18 #define TYPE CUFFT_Z2Z
19 #define EXEC cufftExecZ2Z
20
21 using namespace std;
22
23 //
24 //  Build data structures necessary to pass into kernels
25 //
26
27 struct dataBin {
28   int n, l, m, mtotal, CGsteps, Nsteps, blocksize;
29   REAL dt, epsilon, eta1, eta2, tau, tol, maxR, *Energy_h;
30   dim3 dimBlock, dimGrid;
31   COMPLEX *max_h;
```

```
32 };
33
34 struct operBin {
35   REAL *Lap, *Q;
36 };
37
38 int i;
39
40 //
41 // This series of kernels compartmentalizes the calculation into
      peices that do not depend serially on data.
42 // The kernels are called int the main function below.
43 //
44
45 //
46 // kernE1 and kernE2 are called in the FCHenergy function that
      calculates the FCH energy of the solution
47 //
48
49 __global__ void kernE1(int n, COMPLEX *U, REAL *Lap, REAL *Delx,
      REAL *Dely, REAL *Delz, COMPLEX *LapU, COMPLEX *DelxU, COMPLEX
      *DelyU, COMPLEX *DelzU)
50 {
51   int idx = blockIdx.x*blockDim.x+threadIdx.x;
52   if(idx < n)
53   {
54     U[idx].x = U[idx].x/n;
55     U[idx].y = U[idx].y/n;
56
57     LapU[idx].x = Lap[idx]*U[idx].x;
58     LapU[idx].y = Lap[idx]*U[idx].y;
59
60     DelxU[idx].x = -Delx[idx]*U[idx].y;
61     DelxU[idx].y = Delx[idx]*U[idx].x;
62
63     DelyU[idx].x = -Dely[idx]*U[idx].y;
64     DelyU[idx].y = Dely[idx]*U[idx].x;
65
66     DelzU[idx].x = -Delz[idx]*U[idx].y;
67     DelzU[idx].y = Delz[idx]*U[idx].x;
68
69   }
70 }
71
```

```
72 __global__ void kernE2(int n, REAL epsilon, REAL eta1, REAL eta2,
     REAL tau, REAL *Energy, COMPLEX *U, COMPLEX *LapU, COMPLEX *
     DelxU, COMPLEX *DelyU, COMPLEX *DelzU)
73 {
74   int idx = blockIdx.x*blockDim.x+threadIdx.x;
75   if(idx < n)
76   {
77     Energy[idx] = 0.5*(epsilon*epsilon*LapU[idx].x-(U[idx].x*U[idx
     ].x-1)*(U[idx].x+tau/2))*(epsilon*epsilon*LapU[idx].x-(U[idx].x
     *U[idx].x-1)*(U[idx].x+tau/2))-epsilon*(eta1*epsilon*epsilon
     /2*(DelxU[idx].x*DelxU[idx].x + DelyU[idx].x*DelyU[idx].x +
     DelzU[idx].x*DelzU[idx].x) + eta2*0.5*(U[idx].x+1)*(U[idx].x+1)
     *(0.5*(U[idx].x-1)*(U[idx].x-1)+tau/3*(U[idx].x-2)));
78   }
79 }
80
81 //
82 // kernRes1, kernRes2, kernRes3, and kernRes4 are called in the
     residual function that calculates the residual for the
     conjugate gradient iterations
83 //
84
85 __global__ void kernRes1(int n, COMPLEX *W, COMPLEX *T1, REAL *Lap
     )
86 {
87   int idx = blockIdx.x*blockDim.x+threadIdx.x;
88   if(idx < n)
89   {
90     W[idx].x = W[idx].x/n;
91     W[idx].y = W[idx].y/n;
92
93     T1[idx].x = Lap[idx]*W[idx].x;
94     T1[idx].y = Lap[idx]*W[idx].y;
95   }
96 }
97
98 __global__ void kernRes2(int n, REAL dt, REAL epsilon, REAL eta1,
     REAL eta2, REAL tau, COMPLEX *W, COMPLEX *R, COMPLEX *T1)
99 {
100   int idx = blockIdx.x*blockDim.x+threadIdx.x;
101   if(idx < n)
102   {
103     T1[idx].x = epsilon*epsilon*T1[idx].x - (W[idx].x*W[idx].x-1)
     *(W[idx].x+tau/2);
104     T1[idx].y = 0.0;
```

```
105
106     R[idx].x = -(3*W[idx].x*W[idx].x + tau*W[idx].x - 1 - epsilon*
        eta1)*T1[idx].x + epsilon*(eta1-eta2)*(W[idx].x*W[idx].x-1)*(W[
        idx].x+tau/2);
107     R[idx].y = 0.0;
108   }
109 }
110
111 __global__ void kernRes3(int n, REAL dt, REAL epsilon, COMPLEX *R,
        COMPLEX *T1, REAL *Lap)
112 {
113   int idx = blockIdx.x*blockDim.x+threadIdx.x;
114   if(idx < n)
115   {
116     R[idx].x = dt*Lap[idx]*(epsilon*epsilon*Lap[idx]*T1[idx].x + R
        [idx].x);
117     R[idx].y = dt*Lap[idx]*(epsilon*epsilon*Lap[idx]*T1[idx].y + R
        [idx].y);
118   }
119 }
120
121 __global__ void kernRes4(int n, COMPLEX *W, COMPLEX *U, COMPLEX *R
        )
122 {
123   int idx = blockIdx.x*blockDim.x+threadIdx.x;
124   if(idx < n)
125   {
126     R[idx].x = - W[idx].x + U[idx].x + R[idx].x/n;
127     R[idx].y = 0.0;
128   }
129 }
130
131 //
132 // kernHm1 and kernHm2 are used to compute the H^{-1} inner
        product in the Hm1InnerProd function
133 //
134
135 __global__ void kernHm1(int n, REAL *Lap, COMPLEX *Vhat, COMPLEX *
        innerProd)
136 {
137   int idx = blockIdx.x*blockDim.x+threadIdx.x;
138   if(idx < n && idx != 0)
139   {
140     innerProd[idx].x = Vhat[idx].x/Lap[idx];
141     innerProd[idx].y = Vhat[idx].y/Lap[idx];
```

```
142   }
143   else if (idx == 0)
144   {
145     innerProd[idx].x = 0.0;
146     innerProd[idx].y = 0.0;
147   }
148 }
149
150 __global__ void kernHm2(int n, REAL *Energy, COMPLEX *U, COMPLEX *
      innerProd)
151 {
152   int idx = blockIdx.x*blockDim.x+threadIdx.x;
153   if(idx < n)
154   {
155     Energy[idx] = U[idx].x*innerProd[idx].x/n;
156   }
157 }
158
159 //
160 // kernCG1 through kernCG7 are called in the conjGrad function
161 //
162
163 __global__ void kernCG1(int n, REAL dt, REAL *Q, COMPLEX *Zhat,
      COMPLEX *R, COMPLEX *Phat, COMPLEX *Vhat)
164 {
165   int idx = blockIdx.x*blockDim.x+threadIdx.x;
166   if(idx < n)
167   {
168     Zhat[idx].x = R[idx].x/(1+dt*Q[idx]);
169     Zhat[idx].y = R[idx].y/(1+dt*Q[idx]);
170
171     Phat[idx].x = Zhat[idx].x;
172     Phat[idx].y = Zhat[idx].y;
173
174     Vhat[idx].x = 0.0;
175     Vhat[idx].y = 0.0;
176
177     R[idx].x = R[idx].x/n;
178     R[idx].y = R[idx].y/n;
179   }
180 }
181
182 __global__ void kernCG2(int n, COMPLEX *Phat, COMPLEX *T1, COMPLEX
      *W, COMPLEX *LapU, REAL *Lap)
183 {
```

```
184   int idx = blockIdx.x*blockDim.x+threadIdx.x;
185   if(idx < n)
186   {
187     Phat[idx].x = Phat[idx].x/n;
188     Phat[idx].y = Phat[idx].y/n;
189
190     T1[idx].x = Lap[idx]*Phat[idx].x;
191     T1[idx].y = Lap[idx]*Phat[idx].y;
192
193     W[idx].x = W[idx].x/n;
194     W[idx].y = W[idx].y/n;
195
196     LapU[idx].x = Lap[idx]*W[idx].x;
197     LapU[idx].y = Lap[idx]*W[idx].y;
198   }
199 }
200
201 __global__ void kernCG3(int n, REAL dt, REAL epsilon, REAL eta1,
      REAL eta2, REAL tau, REAL *Lap, COMPLEX *T1, COMPLEX *W,
      COMPLEX *Phat, COMPLEX *JP, COMPLEX *LapU)
202 {
203   int idx = blockIdx.x*blockDim.x+threadIdx.x;
204   if(idx < n)
205   {
206     T1[idx].x = epsilon*epsilon*T1[idx].x - (3*W[idx].x*W[idx].x +
      tau*W[idx].x - 1)*Phat[idx].x;
207     T1[idx].y = 0.0;
208
209     JP[idx].x = -(6*W[idx].x + tau)*Phat[idx].x*(epsilon*epsilon*
    LapU[idx].x - (W[idx].x*W[idx].x-1)*(W[idx].x+tau/2)) - (3*W[
    idx].x*W[idx].x + tau*W[idx].x - 1 - epsilon*eta1)*T1[idx].x +
    epsilon*(eta1-eta2)*(3*W[idx].x*W[idx].x + tau*W[idx].x - 1)*
    Phat[idx].x;
210     JP[idx].y = 0.0;
211   }
212 }
213
214 __global__ void kernCG4(int n, REAL dt, REAL epsilon, COMPLEX *JP,
      COMPLEX *Phat, COMPLEX *T1, REAL *Lap)
215 {
216   int idx = blockIdx.x*blockDim.x+threadIdx.x;
217   if(idx < n)
218   {
219     JP[idx].x = (Phat[idx].x - dt*Lap[idx]*(epsilon*epsilon*Lap[
    idx]*T1[idx].x + JP[idx].x))/n;
```

```
220     JP[idx].y = (Phat[idx].y - dt*Lap[idx]*(epsilon*epsilon*Lap[
    idx]*T1[idx].y + JP[idx].y))/n;
221   }
222 }
223
224 __global__ void kernCG5(int n, REAL alpha, COMPLEX *Vhat, COMPLEX
    *Phat, COMPLEX *R, COMPLEX *JP)
225 {
226   int idx = blockIdx.x*blockDim.x+threadIdx.x;
227   if(idx < n)
228   {
229     Vhat[idx].x = Vhat[idx].x + alpha*Phat[idx].x;
230     Vhat[idx].y = Vhat[idx].y + alpha*Phat[idx].y;
231
232     R[idx].x = R[idx].x - alpha*JP[idx].x;
233     R[idx].y = 0.0;
234   }
235 }
236
237 __global__ void kernCG6(int n, REAL dt, REAL *Q, COMPLEX *Zhat,
    COMPLEX *R)
238 {
239   int idx = blockIdx.x*blockDim.x+threadIdx.x;
240   if(idx < n)
241   {
242     Zhat[idx].x = R[idx].x/(1+dt*Q[idx]);
243     Zhat[idx].y = R[idx].y/(1+dt*Q[idx]);
244
245     R[idx].x = R[idx].x/n;
246     R[idx].y = R[idx].y/n;
247   }
248 }
249
250 __global__ void kernCG7(int n, REAL beta1, REAL beta2, COMPLEX *
    Phat, COMPLEX *Zhat)
251 {
252   int idx = blockIdx.x*blockDim.x+threadIdx.x;
253   if(idx < n)
254   {
255     Phat[idx].x = Zhat[idx].x + beta2/beta1 *Phat[idx].x;
256     Phat[idx].y = Zhat[idx].y + beta2/beta1 *Phat[idx].y;
257   }
258 }
259
260 //
```

```
261 // kernN1 is called in the Newton function
262 //
263
264 __global__ void kernN1(int n, COMPLEX *Vhat, COMPLEX *W)
265 {
266   int idx = blockIdx.x*blockDim.x+threadIdx.x;
267   if(idx < n)
268   {
269     Vhat[idx].x = Vhat[idx].x/n;
270
271     W[idx].x += Vhat[idx].x;
272     W[idx].y = 0.0;
273   }
274 }
275
276 __global__ void kernLapQ(int n, REAL epsilon, REAL c1, REAL c3,
      REAL *Lap, REAL *Q, REAL *Delx, REAL *Dely, REAL *Delz)
277 {
278   int idx = blockIdx.x*blockDim.x+threadIdx.x;
279   if(idx < n)
280   {
281     Lap[idx] = -(Delx[idx]*Delx[idx]+Dely[idx]*Dely[idx]+Delz[idx
      ]*Delz[idx]);
282     Q[idx] = -Lap[idx]*(epsilon*epsilon*epsilon*epsilon*Lap[idx]*
      Lap[idx] - c1*epsilon*epsilon*Lap[idx] - c3);
283   }
284 }
285
286 //
287 // kern1 through kern5 are called in the main function
288 //
289
290 __global__ void kern1(int n, REAL *Lap, COMPLEX *U, COMPLEX *LapU)
291 {
292   int idx = blockIdx.x*blockDim.x+threadIdx.x;
293   if(idx < n)
294   {
295     U[idx].x = U[idx].x/n;
296     U[idx].y = U[idx].y/n;
297
298     LapU[idx].x = Lap[idx]*U[idx].x;
299     LapU[idx].y = Lap[idx]*U[idx].y;
300   }
301 }
302
```

```
303 __global__ void kern2(int n, REAL epsilon, REAL eta1, REAL eta2,
    REAL tau, COMPLEX *LapU, COMPLEX *T1, COMPLEX *U, COMPLEX *Wexp
    )
304 {
305   int idx = blockIdx.x*blockDim.x+threadIdx.x;
306   if(idx < n)
307   {
308     T1[idx].x = epsilon*epsilon*LapU[idx].x - (U[idx].x*U[idx].x -
      1)*(U[idx].x + tau/2);
309     T1[idx].y = 0.0;
310
311     Wexp[idx].x = -(3*U[idx].x*U[idx].x + tau*U[idx].x - 1 -
    epsilon*eta1)*T1[idx].x + epsilon*(eta1-eta2)*(U[idx].x*U[idx].
    x - 1)*(U[idx].x + tau/2);
312     Wexp[idx].y = 0.0;
313   }
314 }
315
316 __global__ void kern3(int n, REAL dt, REAL epsilon, REAL *Lap,
    COMPLEX *Wexp, COMPLEX *T1)
317 {
318   int idx = blockIdx.x*blockDim.x+threadIdx.x;
319   if(idx < n)
320   {
321     Wexp[idx].x = dt*Lap[idx]*(epsilon*epsilon*Lap[idx]*T1[idx].x
    + Wexp[idx].x);
322     Wexp[idx].y = dt*Lap[idx]*(epsilon*epsilon*Lap[idx]*T1[idx].y
    + Wexp[idx].y);
323   }
324 }
325
326 __global__ void kern4(int n, COMPLEX *Wexp, COMPLEX *U, COMPLEX *W
    )
327 {
328   int idx = blockIdx.x*blockDim.x+threadIdx.x;
329   if(idx < n)
330   {
331     Wexp[idx].x = U[idx].x + Wexp[idx].x/n;
332     Wexp[idx].y = U[idx].y + Wexp[idx].y/n;
333
334     W[idx].x = Wexp[idx].x;
335     W[idx].y = Wexp[idx].y;
336   }
337 }
338
```

```
339 //
340 // kernError is to compute the error for the adaptive time
       stepping scheme
341 //
342
343 __global__ void kernError(int n, COMPLEX *R, COMPLEX *Wexp,
       COMPLEX *W)
344 {
345   int idx = blockIdx.x*blockDim.x+threadIdx.x;
346   if(idx < n)
347   {
348     R[idx].x = 0.5*fabs(Wexp[idx].x - W[idx].x);
349   }
350 }
351
352 __global__ void kern5(int n, COMPLEX *U, COMPLEX *W)
353 {
354   int idx = blockIdx.x*blockDim.x+threadIdx.x;
355   if(idx < n)
356   {
357       U[idx].x = W[idx].x;
358       U[idx].y = 0.0; //W[idx].y;
359   }
360 }
361
362 //
363 // warpReduce, reduce, warpReduceSum, and reduceSum parallelize
       the reductions necessary to calculate energy and error
364 //
365
366 __device__ void warpReduce(volatile REAL *sdata, unsigned int tid,
       int blockSize)
367 {
368   if (blockSize >= 64) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 32]));
369   if (blockSize >= 32) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 16]));
370   if (blockSize >= 16) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 8]));
371   if (blockSize >= 8) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 4]));
372   if (blockSize >= 4) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 2]));
373   if (blockSize >= 2) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 1]));
```

```
374 }
375
376 __global__ void reduce(COMPLEX *g_idata, COMPLEX *g_odata, int n,
      int blockSize)
377 {
378   extern __shared__ REAL sdata[];
379   unsigned int tid = threadIdx.x;
380   unsigned int i = blockIdx.x*(blockSize*2) + tid;
381   unsigned int gridSize = blockSize*2*gridDim.x;
382   sdata[tid] = 0;
383
384   while (i < n) {sdata[tid] = max(fabs(g_idata[i].x),fabs(g_idata[
       i+blockSize].x)); i += gridSize; }
385   __syncthreads();
386
387   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 256])); } __syncthreads(); }
388   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 128])); } __syncthreads(); }
389   if (blockSize >= 128) { if (tid < 64) { sdata[tid] = max(fabs(
       sdata[tid]), fabs(sdata[tid + 64])); } __syncthreads(); }
390   if (tid < 32) warpReduce(sdata, tid, blockSize);
391   if (tid == 0) g_odata[blockIdx.x].x = sdata[0];
392 }
393
394 __device__ void warpReduceSum(volatile REAL *sdata, unsigned int
      tid, int blockSize)
395 {
396   if (blockSize >= 64) sdata[tid] = sdata[tid] + sdata[tid + 32];
397   if (blockSize >= 32) sdata[tid] = sdata[tid] + sdata[tid + 16];
398   if (blockSize >= 16) sdata[tid] = sdata[tid] + sdata[tid + 8];
399   if (blockSize >= 8) sdata[tid] = sdata[tid] + sdata[tid + 4];
400   if (blockSize >= 4) sdata[tid] = sdata[tid] + sdata[tid + 2];
401   if (blockSize >= 2) sdata[tid] = sdata[tid] + sdata[tid + 1];
402 }
403
404 __global__ void reduceSum(REAL *g_idata, REAL *g_odata, int n, int
       blockSize)
405 {
406   extern __shared__ REAL sdata[];
407   unsigned int tid = threadIdx.x;
408   unsigned int i = blockIdx.x*(blockSize*2) + tid;
409   unsigned int gridSize = blockSize*2*gridDim.x;
410   sdata[tid] = 0;
411
```

```
412   while (i < n) {sdata[tid] = g_idata[i] + g_idata[i+blockSize]; i
        += gridSize; }
413   __syncthreads();
414
415   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = sdata[tid]
        + sdata[tid + 256]; } __syncthreads(); }
416   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = sdata[tid]
        + sdata[tid + 128]; } __syncthreads(); }
417   if (blockSize >= 128) { if (tid < 64) { sdata[tid] = sdata[tid]
       + sdata[tid + 64]; } __syncthreads(); }
418   if (tid < 32) warpReduceSum(sdata, tid, blockSize);
419   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
420 }
421
422 //
423 // FCHenergy computes the energy of the solution U
424 //
425
426 REAL FCHenergy(dataBin &data, operBin oper, COMPLEX *U, REAL *
       Energy, REAL *Delx, REAL *Dely, REAL *Delz, COMPLEX *LapU,
       COMPLEX *DelxU, COMPLEX *DelyU, COMPLEX *DelzU, cufftHandle
       plan) {
427
428   EXEC (plan, U, U, CUFFT_FORWARD);
429
430   kernE1<<< data.dimGrid, data.dimBlock>>>(data.n, U, oper.Lap,
       Delx, Dely, Delz, LapU, DelxU, DelyU, DelzU);
431
432   EXEC(plan, LapU, LapU, CUFFT_INVERSE);
433   EXEC(plan, DelxU, DelxU, CUFFT_INVERSE);
434   EXEC(plan, DelyU, DelyU, CUFFT_INVERSE);
435   EXEC(plan, DelzU, DelzU, CUFFT_INVERSE);
436   EXEC(plan, U, U, CUFFT_INVERSE);
437
438   kernE2<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
       data.eta1, data.eta2, data.tau, Energy, U, LapU, DelxU, DelyU,
       DelzU);
439
440   reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
       REAL)>>>(Energy, Energy, data.n, data.blocksize);
441
442   int nRed = data.n/(data.blocksize*2);
443   while (nRed > 1)
444   {
```

```
445     reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*
        sizeof(REAL)>>>(Energy, Energy, nRed, data.blocksize);
446     nRed = nRed/(data.blocksize*2);
447   }
448
449   cudaMemcpy(data.Energy_h, Energy, sizeof(REAL),
        cudaMemcpyDeviceToHost);
450 //  tEtail->energy = *data.Energy_h;
451
452   return(*data.Energy_h);
453 }
454
455 //
456 // residual computes the residual needed in the standard conjugate
        gradient algorithm used below
457 //
458
459 void residual (dataBin &data, operBin oper, COMPLEX *U, COMPLEX *W
      , COMPLEX *R, COMPLEX *T1, cufftHandle plan) {
460
461   EXEC(plan, W, W, CUFFT_FORWARD);
462
463   kernRes1<<< data.dimGrid, data.dimBlock>>>(data.n, W, T1, oper.
        Lap);
464
465   EXEC(plan, T1, T1, CUFFT_INVERSE);
466   EXEC(plan, W, W, CUFFT_INVERSE);
467
468   kernRes2<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, data
        .epsilon, data.eta1, data.eta2, data.tau, W, R, T1);
469
470   EXEC(plan, T1, T1, CUFFT_FORWARD);
471   EXEC(plan, R, R, CUFFT_FORWARD);
472
473   kernRes3<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, data
        .epsilon, R, T1, oper.Lap);
474
475   EXEC(plan, R, R, CUFFT_INVERSE);
476
477   kernRes4<<< data.dimGrid, data.dimBlock>>>(data.n, W, U, R);
478 }
479
480 //
481 // Hm1InnerProd computes the inner product in the H^{-1} norm
        between U and V
```

```
482 //
483
484 REAL Hm1InnerProd(dataBin &data, operBin oper, COMPLEX *U, COMPLEX
        *Vhat, COMPLEX *innerProd, REAL *Energy, cufftHandle plan) {
485
486   kernHm1<<< data.dimGrid, data.dimBlock>>>(data.n, oper.Lap, Vhat
       , innerProd);
487
488   EXEC(plan, innerProd, innerProd, CUFFT_INVERSE);
489
490   kernHm2<<< data.dimGrid, data.dimBlock>>>(data.n, Energy, U,
       innerProd);
491
492   reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
       REAL)>>>(Energy, Energy, data.n, data.blocksize);
493
494   int nRed = data.n/(data.blocksize*2);
495   while (nRed > 1)
496   {
497     reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*
       sizeof(REAL)>>>(Energy, Energy, nRed, data.blocksize);
498     nRed = nRed/(data.blocksize*2);
499   }
500
501   cudaMemcpy(data.Energy_h, Energy, sizeof(REAL),
       cudaMemcpyDeviceToHost);
502 //   tEtail->energy = *data.Energy_h;
503
504   return(*data.Energy_h);
505 }
506
507 //
508 // conjGrad executes the standard conjugate gradient scheme
509 //
510
511 void conjGrad(dataBin &data, operBin oper, COMPLEX *U, COMPLEX *W,
       COMPLEX *R, COMPLEX *T1, COMPLEX *LapU, COMPLEX *Zhat, COMPLEX
       *Phat, COMPLEX *Vhat, COMPLEX *JP, COMPLEX *innerProd, REAL *
       Energy, COMPLEX *max, cufftHandle plan) {
512
513   REAL alpha, beta1, beta2;
514
515   residual (data, oper, U, W, R, T1, plan);
516
517   EXEC(plan, R, R, CUFFT_FORWARD);
```

```
518
519   kernCG1<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, oper.
      Q, Zhat, R, Phat, Vhat);
520
521   EXEC(plan, R, R, CUFFT_INVERSE);
522
523   data.m = 0;
524   beta1 = Hm1InnerProd(data, oper, R, Zhat, innerProd, Energy,
      plan);
525
526   while (data.m <= data.CGsteps) {
527
528     EXEC(plan, W, W, CUFFT_FORWARD);
529
530     kernCG2<<< data.dimGrid, data.dimBlock>>>(data.n, Phat, T1, W,
      LapU, oper.Lap);
531
532     EXEC(plan, Phat, Phat, CUFFT_INVERSE);
533     EXEC(plan, T1, T1, CUFFT_INVERSE);
534     EXEC(plan, W, W, CUFFT_INVERSE);
535     EXEC(plan, LapU, LapU, CUFFT_INVERSE);
536
537     kernCG3<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt,
      data.epsilon, data.eta1, data.eta2, data.tau, oper.Lap, T1, W,
      Phat, JP, LapU);
538
539     EXEC(plan, Phat, Phat, CUFFT_FORWARD);
540     EXEC(plan, JP, JP, CUFFT_FORWARD);
541     EXEC(plan, T1, T1, CUFFT_FORWARD);
542
543     kernCG4<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt,
      data.epsilon, JP, Phat, T1, oper.Lap);
544
545     EXEC(plan, JP, JP, CUFFT_INVERSE);
546
547     alpha = Hm1InnerProd(data, oper, JP, Phat, innerProd, Energy,
      plan);
548
549     alpha = beta1/alpha;
550
551     kernCG5<<< data.dimGrid, data.dimBlock>>>(data.n, alpha, Vhat,
      Phat, R, JP);
552
553     data.m++;
554
```

```
555     reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
        REAL)>>>(R, max, data.n, data.blocksize);
556     int nRed = data.n/(data.blocksize*2);
557     while (nRed > 1)
558     {
559       reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof
        (REAL)>>>(max, max, nRed, data.blocksize);
560       nRed = nRed/(data.blocksize*2);
561     }
562
563     cudaMemcpy(data.max_h, max, sizeof(REAL),
        cudaMemcpyDeviceToHost);
564
565     data.maxR = data.max_h->x;
566
567     if (data.maxR < data.tol/10) {
568       break;
569     }
570
571     EXEC(plan, R, R, CUFFT_FORWARD);
572
573     kernCG6<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt,
        oper.Q, Zhat, R);
574
575     EXEC(plan, R, R, CUFFT_INVERSE);
576
577     beta2 = Hm1InnerProd(data, oper, R, Zhat, innerProd, Energy,
        plan);
578
579     kernCG7<<< data.dimGrid, data.dimBlock>>>(data.n, beta1, beta2
        , Phat, Zhat);
580
581     beta1 = beta2;
582   }
583
584 }
585
586 //
587 // newton executes Newton's method for finding zeros of the
        function and depends on the conjGrad function above
588 //
589
590 void newton(dataBin &data, operBin oper, COMPLEX *U, COMPLEX *W,
        COMPLEX *Vhat, COMPLEX *R, COMPLEX *T1, COMPLEX *LapU, COMPLEX
        *Zhat, COMPLEX *Phat, COMPLEX *JP, COMPLEX *innerProd, REAL *
```

```
    Energy, COMPLEX *max, cufftHandle plan) {
591   while (data.l <= data.Nsteps) {
592     conjGrad(data, oper, U, W, R, T1, LapU, Zhat, Phat, Vhat, JP,
      innerProd, Energy, max, plan);
593
594     data.mtotal +=data.m;
595     if (data.m >= data.CGsteps) {break;}
596     data.l++;
597
598     EXEC(plan, Vhat, Vhat, CUFFT_INVERSE);
599
600     kernN1<<< data.dimGrid, data.dimBlock>>>(data.n, Vhat, W);
601
602     residual (data, oper, U, W, R, T1, plan);
603
604     reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
      REAL)>>>(R, max, data.n, data.blocksize);
605     int nRed = data.n/(data.blocksize*2);
606     while (nRed > 1)
607     {
608       reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof
      (REAL)>>>(max, max, nRed, data.blocksize);
609       nRed = nRed/(data.blocksize*2);
610     }
611
612     cudaMemcpy(data.max_h, max, sizeof(REAL),
      cudaMemcpyDeviceToHost);
613
614     data.maxR = data.max_h->x;
615     if (data.maxR < data.tol) {break;}
616   }
617 }
618
619 //
620 // The node structure builds a linked list that collects energy
      and time data
621 //
622
623 struct node {
624   REAL ctime;
625   REAL energy;
626   int mtotal;
627   struct node* next;
628 };
629
```

```
630 void addNode(struct node*& tail, REAL ctime) {
631   struct node* newNode = new node;
632   newNode->ctime = ctime;
633   newNode->energy = 0.0;
634   newNode->mtotal = 0;
635   newNode->next = NULL;
636   tail->next = newNode;
637   tail = newNode;
638   }
639
640 //
641 // This is where the main function begins. To make sense of the
      code, begin here.
642 //
643
644 int main (int argc, char * const argv[]) {
645   if (argc <=1 )
646   {
647     cout << "Usage: " << argv[0] << " <filename> <output off/
      increasing/even (0/1/2)>" << endl;
648     exit(1);
649   }
650
651   time_t tic, toc;
652   time_t tic2, toc2;
653   time(&tic);
654   //
655   // Initialize necessary constants
656   //
657   cout << scientific;
658   cout.precision(10);
659   int i, j, k;
660
661   fstream myfile, myfile2, myfile3, myfile4;
662   stringstream sstm;
663   char datain[50];
664
665   REAL ctime, Tmax, Tpicstep, Lx, Ly, Lz, c1, c2, c3, Ttol,
      Tfactor, Error;
666   int Nx, Ny, Nz, Mx, My, Mz;
667   bool CGflag = true;
668
669   REAL tPic = 1.0;
670   struct node* tEhead = NULL;
671   struct node* tEtail = NULL;
```

```
672
673   tEhead = new node;
674   tEhead->ctime = 0.0;
675   tEhead->energy = 0.0;
676   tEtail = tEhead;
677
678   string filein, fileUout, filetEout;
679
680   dataBin data;
681   operBin oper;
682
683   //
684   // Read in all the necessary data from the file given by the
       user
685   //
686
687   myfile3.open(argv[1], ios::in);
688
689   if(myfile3.is_open())
690   {
691     myfile3.ignore(512,'=');
692     myfile3 >> datain;
693     filein = datain;
694     cout << "Init cond file = " << filein << endl;
695
696     myfile3.ignore(512,'=');
697     myfile3 >> datain;
698     fileUout = datain;
699     cout << "Write out solution file = " << fileUout << ".dat" <<
       endl;
700
701     myfile3.ignore(512,'=');
702     myfile3 >> datain;
703     filetEout = datain;
704     cout << "Write out time and energy file = " << filetEout << ".
       dat" << endl;
705
706     myfile3.ignore(512,'=');
707     myfile3 >> datain;
708     data.epsilon = atof(datain);
709     cout << "epsilon = " << data.epsilon << endl;
710
711     myfile3.ignore(512,'=');
712     myfile3 >> datain;
713     data.eta1 = atof(datain);
```

```
714     cout << "eta1 = " << data.eta1 << endl;
715
716     myfile3.ignore(512,'=');
717     myfile3 >> datain;
718     data.eta2 = atof(datain);
719     cout << "eta2 = " << data.eta2 << endl;
720
721     myfile3.ignore(512,'=');
722     myfile3 >> datain;
723     data.tau = atof(datain);
724     cout << "tau = " << data.tau << endl;
725
726     myfile3.ignore(512,'=');
727     myfile3 >> datain;
728     data.dt = atof(datain);
729     cout << "dt = " << data.dt << endl;
730
731     myfile3.ignore(512,'=');
732     myfile3 >> datain;
733     Tmax = atof(datain);
734     cout << "Tmax = " << Tmax << endl;
735
736     myfile3.ignore(512,'=');
737     myfile3 >> datain;
738     Tpicstep = atof(datain);
739     cout << "Tpicstep = " << Tpicstep << endl;
740
741     myfile3.ignore(512,'=');
742     myfile3 >> datain;
743     Nx = atoi(datain);
744     cout << "Nx = " << Nx << endl;
745
746     myfile3.ignore(512,'=');
747     myfile3 >> datain;
748     Ny = atoi(datain);
749     cout << "Ny = " << Ny << endl;
750
751     myfile3.ignore(512,'=');
752     myfile3 >> datain;
753     Nz = atoi(datain);
754     cout << "Nz = " << Nz << endl;
755
756     myfile3.ignore(512,'=');
757     myfile3 >> datain;
758     Lx = atof(datain);
```

```
759     cout << "Lx = " << Lx << endl;
760
761     myfile3.ignore(512,'=');
762     myfile3 >> datain;
763     Ly = atof(datain);
764     cout << "Ly = " << Ly << endl;
765
766     myfile3.ignore(512,'=');
767     myfile3 >> datain;
768     Lz = atof(datain);
769     cout << "Lz = " << Lz << endl;
770
771     myfile3.ignore(512,'=');
772     myfile3 >> datain;
773     c1 = atof(datain);
774     cout << "c1 = " << c1 << endl;
775
776     myfile3.ignore(512,'=');
777     myfile3 >> datain;
778     c2 = atof(datain);
779     cout << "c2 = " << c2 << endl;
780
781     myfile3.ignore(512,'=');
782     myfile3 >> datain;
783     c3 = atof(datain);
784     cout << "c3 = " << c3 << endl;
785
786     myfile3.ignore(512,'=');
787     myfile3 >> datain;
788     data.CGsteps = atof(datain);
789     cout << "CGsteps = " << data.CGsteps << endl;
790
791     myfile3.ignore(512,'=');
792     myfile3 >> datain;
793     data.Nsteps = atof(datain);
794     cout << "Nsteps = " << data.Nsteps << endl;
795
796     myfile3.ignore(512,'=');
797     myfile3 >> datain;
798     data.tol = atof(datain);
799     cout << "tol = " << data.tol << endl;
800
801     myfile3.ignore(512,'=');
802     myfile3 >> datain;
803     Ttol = atof(datain);
```

```
804       cout << "Ttol = " << Ttol << endl;
805
806       myfile3.ignore(512,'=');
807       myfile3 >> datain;
808       Tfactor = atof(datain);
809       cout << "Tfactor = " << Tfactor << endl;
810
811       myfile3.ignore(512,'=');
812       myfile3 >> datain;
813       data.blocksize = atoi(datain);
814       cout << "Block size = " << data.blocksize << endl;
815       cout << "Grid size = " << Nx*Ny*Nz/data.blocksize << endl;
816     }
817     else { cout << "File could not be read." << endl; }
818
819     ctime = 0.0;
820
821     if (*argv[2] == '2')
822     {
823       tPic = Tpicstep;
824     }
825
826     Mx = Nx/2;
827     My = Ny/2;
828     Mz = Nz/2;
829     data.n = Nx*Ny*Nz;
830     data.dimBlock = data.blocksize;
831     data.dimGrid = data.n/data.dimBlock.x;
832
833
834     //
835     // Build the laplacian array, and dynamically allocate other
        needed arrays
836     //
837
838     REAL *xkvec, *ykvec, *zkvec;
839     xkvec = new REAL[Nx];
840     ykvec = new REAL[Ny];
841     zkvec = new REAL[Nz];
842
843     for (i=0; i<Mx; i++)
844     {
845       xkvec[i] = i*PI/Lx;
846       xkvec[i+Mx] = (i-Mx)*PI/Lx;
847     }
```

```
848
849   for (i=0; i<My; i++)
850   {
851     ykvec[i] = i*PI/Ly;
852     ykvec[i+My] = (i-My)*PI/Ly;
853   }
854
855   for (i=0; i<Mz; i++)
856   {
857     zkvec[i] = i*PI/Lz;
858     zkvec[i+Mz] = (i-Mz)*PI/Lx;
859   }
860
861   REAL *Delx_h, *Dely_h, *Delz_h, *Delx, *Dely, *Delz, *Energy;
862   data.Energy_h = new REAL;
863   Delx_h = new REAL[data.n];
864   Dely_h = new REAL[data.n];
865   Delz_h = new REAL[data.n];
866
867   for (i=0; i<Nx; i++) {
868     for (j=0; j<Ny; j++) {
869       for (k=0; k<Nz; k++)
870       {
871         Delx_h[k+Nz*(j+Ny*i)] = xkvec[i];
872         Dely_h[k+Nz*(j+Ny*i)] = ykvec[j];
873         Delz_h[k+Nz*(j+Ny*i)] = zkvec[k];
874       }
875     }
876   }
877
878   cout << "Del operators built" << endl;
879
880   //
881   // Initialize all of the arrays that will be used in fourier
       transforms, and build the plans necessary
882   //
883
884   int nBytes = sizeof(COMPLEX)*data.n;
885   cout << "nBytes is = " << nBytes << endl;
886   COMPLEX *U_h, *U, *W, *Wexp, *Vhat, *R, *T1, *LapU, *Zhat, *Phat
       , *JP, *DelxU, *DelyU, *DelzU, *innerProd, *max;
887
888   //
889   // Read in inital U data from filein
890   //
```

```
891
892   U_h = new COMPLEX[nBytes];
893   data.max_h = new COMPLEX;
894
895   myfile.open(filein.c_str(),ios::in);
896
897   if(myfile.is_open())
898   {
899     for (i=0; i<data.n; i++)
900     {
901       myfile.getline(datain, 50);
902       U_h[i].x = atof(datain);
903       U_h[i].y = 0.0;
904     }
905     cout << "Initial data read in successfully" << endl;
906   } else { cout << "Initial data not read" << endl; }
907
908   myfile.close();
909
910   //
911   // Allocate memory on the GPU and set up space for the Fourier
       transforms
912   //
913
914   cudaMalloc((void **)&U, nBytes);
915   cudaMalloc((void **)&W, nBytes);
916   cudaMalloc((void **)&Wexp, nBytes);
917   cudaMalloc((void **)&Vhat, nBytes);
918   cudaMalloc((void **)&R, nBytes);
919   cudaMalloc((void **)&T1, nBytes);
920   cudaMalloc((void **)&LapU, nBytes);
921   cudaMalloc((void **)&Zhat, nBytes);
922   cudaMalloc((void **)&Phat, nBytes);
923   cudaMalloc((void **)&JP, nBytes);
924   cudaMalloc((void **)&DelxU, nBytes);
925   cudaMalloc((void **)&DelyU, nBytes);
926   cudaMalloc((void **)&DelzU, nBytes);
927   cudaMalloc((void **)&innerProd, nBytes);
928   cudaMalloc((void **)&Energy, sizeof(REAL)*data.n);
929
930   cudaMalloc((void **)&oper.Lap, sizeof(REAL)*data.n);
931   cudaMalloc((void **)&oper.Q, sizeof(REAL)*data.n);
932   cudaMalloc((void **)&Delx, sizeof(REAL)*data.n);
933   cudaMalloc((void **)&Dely, sizeof(REAL)*data.n);
934   cudaMalloc((void **)&Delz, sizeof(REAL)*data.n);
```

```
935   cudaMalloc((void **)&max, sizeof(COMPLEX)*data.dimGrid.x/2);
936
937   cufftHandle plan;
938   cufftPlan3d(&plan, Nx, Ny, Nz, TYPE);
939
940
941   cudaMemcpy(Delx, Delx_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
942   cudaMemcpy(Dely, Dely_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
943   cudaMemcpy(Delz, Delz_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
944   cudaMemcpy(U, U_h, nBytes, cudaMemcpyHostToDevice);
945   cout << "Initial data copied" << endl;
946
947   //
948   // Calculate the constant arrays Lap and Q that will be used
       below
949   //
950
951   kernLapQ<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
        c1, c3, oper.Lap, oper.Q, Delx, Dely, Delz);
952   cout << "Lap and Q initialized" << endl;
953
954   //
955   // while loop governing time stepping
956   //
957   while (ctime < Tmax)
958   {
959     time(&tic2);
960
961     //
962     // Prepare the arrays needed to use CG based Newton's method
963     //
964
965     EXEC(plan, U, U, CUFFT_FORWARD);
966
967     kern1<<< data.dimGrid, data.dimBlock>>>(data.n, oper.Lap, U,
       LapU);
968
969     EXEC(plan, LapU, LapU, CUFFT_INVERSE);
970     EXEC(plan, U, U, CUFFT_INVERSE);
971
972     tEtail->energy = FCHenergy(data, oper, U, Energy, Delx, Dely,
       Delz, LapU, DelxU, DelyU, DelzU, plan);
```

```
973
974     kern2<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
        data.eta1, data.eta2, data.tau, LapU, T1, U, Wexp);
975
976     EXEC(plan, Wexp, Wexp, CUFFT_FORWARD);
977     EXEC(plan, T1, T1, CUFFT_FORWARD);
978
979     kern3<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, data.
        epsilon, oper.Lap, Wexp, T1);
980
981     EXEC(plan, Wexp, Wexp, CUFFT_INVERSE);
982
983     kern4<<< data.dimGrid, data.dimBlock>>>(data.n, Wexp, U, W);
984
985     data.l = 0;
986     data.m = 0;
987     data.mtotal = 0;
988
989     //
990     // Apply Newton's method
991     //
992
993     newton(data, oper, U, W, Vhat, R, T1, LapU, Zhat, Phat, JP,
        innerProd, Energy, max, plan);
994
995     if (data.m >= data.CGsteps) {cout << "No Conjugate Gradient
        convergence at time t = " << tEtail-> ctime << endl;}
996
997     if (data.l >= data.Nsteps) {cout << "No Newton convergence at
        time t = " << tEtail-> ctime << endl;}
998
999     kernError<<< data.dimGrid, data.dimBlock>>>(data.n, R, Wexp, W
        );
1000
1001    reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
        REAL)>>>(R, max, data.n, data.blocksize);
1002    int nRed = data.n/(data.blocksize*2);
1003    while (nRed > 1)
1004    {
1005      reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof
        (REAL)>>>(max, max, nRed, data.blocksize);
1006      nRed = nRed/(data.blocksize*2);
1007    }
1008
```

```
1009      cudaMemcpy(data.max_h, max, sizeof(REAL),
          cudaMemcpyDeviceToHost);
1010
1011      Error = data.max_h->x;
1012
1013      //
1014      // Check to see if the iterations converged then adjust the
          time step accordingly
1015      //
1016
1017      if (Error > Ttol || data.m >= data.CGsteps || data.l >= data.
          Nsteps) {
1018         cout << "Tolerance fail! Recomputing " << Error << "\t l = "
           << data.l << "\t m = " << data.m << endl;
1019         data.dt = data.dt/1.3;
1020      }
1021      else {
1022         ctime = ctime + data.dt;
1023         tEtail->mtotal = data.mtotal;
1024
1025         //
1026         // Update time and update U to be the solution at the next
          time step
1027         //
1028
1029         kern5<<< data.dimGrid, data.dimBlock>>>(data.n, U, W);
1030
1031         time(&toc2);
1032         cout << ctime << "\t" << data.dt << "\t" << tEtail->energy
          << "\t" << data.l << "\t" << data.m << "\t" << data.mtotal << "
          \t" << difftime(toc2,tic2) << endl;
1033
1034         if (data.mtotal < data.CGsteps/10 && data.l < data.Nsteps/3)
           {
1035            if (CGflag == true) {
1036               data.dt = data.dt*min(Tfactor*sqrt(Ttol/Error), (REAL)
          1.3);
1037            }
1038            else {
1039               data.dt = data.dt*min(Tfactor*sqrt(Ttol/Error), (REAL)
          (1.0 + 0.003*(data.CGsteps/10-data.mtotal)));
1040            }
1041         }
1042         else
1043         {
```

```
1044          data.dt = data.dt*min(Tfactor*sqrt(Ttol/Error), (REAL)
      1.0);
1045          CGflag = false;
1046        }
1047
1048        //
1049        // Write out solution to hard disk if required
1050        //
1051
1052        if (abs(ctime - tPic) < data.tol/100 && *argv[2] != '0')
1053        {
1054          cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
1055
1056          sstm << fileUout << (int) tPic << ".dat";
1057          myfile2.open(sstm.str().c_str(),ios::out);
1058          if (myfile2.is_open())
1059          {
1060
1061            for (i=0; i<data.n; i++)
1062            {
1063              myfile2 << U_h[i].x << endl;
1064            }
1065
1066          }
1067          cout << "Writing out " << sstm.str() << endl;
1068          myfile2.close();
1069          sstm.str("");
1070
1071          if (*argv[2] == '1')
1072          {
1073            tPic += pow((REAL) 10.0,(REAL) ceil(log10(tPic+1))-1);
1074          }
1075          else
1076          {
1077            tPic += Tpicstep;
1078          }
1079        }
1080        //
1081        // Adjust the timestep if necessary
1082        //
1083        if (data.dt > (Tmax - ctime))
1084        {
1085          data.dt = Tmax - ctime;
1086        }
1087
```

```
1088        if (data.dt > (tPic - ctime) && *argv[2] != '0')
1089        {
1090          data.dt = tPic - ctime;
1091        }
1092
1093        addNode(tEtail, ctime);
1094
1095      }
1096    }
1097
1098    tEtail->energy = FCHenergy(data, oper, W, Energy, Delx, Dely,
       Delz, LapU, DelxU, DelyU, DelzU, plan);
1099
1100    //
1101    // Write final solution to hard disk and compute its energy
1102    //
1103    sstm << fileUout << (int) Tmax << ".dat";
1104    myfile2.open(sstm.str().c_str(),ios::out);
1105    if (myfile2.is_open())
1106    {
1107      cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
1108
1109      myfile2.precision(15);
1110      for (i=0; i<data.n; i++)
1111      {
1112        myfile2 << U_h[i].x << endl;
1113      }
1114
1115    }
1116    cout << "Writing out " << sstm.str() << endl;
1117    myfile2.close();
1118    sstm.str("");
1119
1120    sstm << filetEout << ".dat";
1121    myfile4.open(sstm.str().c_str(),ios::out);
1122    if (myfile4.is_open())
1123    {
1124
1125      while (tEhead != NULL)
1126      {
1127        myfile4 << tEhead->ctime << "\t" << tEhead->energy << "\t"
       << tEhead->mtotal << endl;
1128        tEtail = tEhead;
1129        tEhead = tEhead->next;
1130        delete tEtail;     }
```

```
1131
1132    }
1133    myfile4.close();
1134    delete tEhead;
1135
1136    //
1137    // Perform clean up of memory used
1138    //
1139
1140    cudaFree(U);
1141    cudaFree(U_h);
1142    cudaFree(W);
1143    cudaFree(Wexp);
1144    cudaFree(Vhat);
1145    cudaFree(R);
1146    cudaFree(T1);
1147    cudaFree(LapU);
1148    cudaFree(Zhat);
1149    cudaFree(Phat);
1150    cudaFree(JP);
1151    cudaFree(DelxU);
1152    cudaFree(DelyU);
1153    cudaFree(DelzU);
1154    cudaFree(innerProd);
1155    cudaFree(oper.Lap);
1156    cudaFree(oper.Q);
1157    cudaFree(max);
1158
1159    cufftDestroy(plan);
1160
1161    delete data.Energy_h;
1162    delete data.max_h;
1163    delete[] U_h;
1164    delete[] xkvec;
1165    delete[] ykvec;
1166    delete[] zkvec;
1167    delete[] Delx_h;
1168    delete[] Dely_h;
1169    delete[] Delz_h;
1170
1171    time(&toc);
1172    cout << "\n" << difftime(toc,tic) << endl;
1173
1174      return 0;
1175 }
```

./code/FCHImp.cu

Appendix C: ETD-RK2 Code in CUDA

```
1  //
2  // FCHExInt.cu
3  //
4  // This code was written by Jaylan Jones to approximate solutions
       to the
5  // Functionalized Cahn-Hilliard Equation using the Exponential
       Time
6  // Differencing Runge-Kutta scheme that is second order accurate
       in time.
7  //
8
9  #include <iostream>
10 #include <fstream>
11 #include <sstream>
12 #include <cuda.h>
13 #include <cufft.h>
14 #include <time.h>
15 #include <math.h>
16
17 #define PI 3.141592653589
18 #define REAL double
19 #define COMPLEX cufftDoubleComplex
20 #define TYPE CUFFT_Z2Z
21 #define EXEC cufftExecZ2Z
22
23 using namespace std;
24
25 //
26 //  Build data structures necessary to pass into kernels
27 //
28
29 struct dataBin {
30   int n, blocksize;
```

```
31   REAL dt, epsilon, eta1, eta2, tau, mu, maxR, *Energy_h;
32   dim3 dimBlock, dimGrid;
33   COMPLEX *max_h, *zf;
34 };
35
36 struct operBin {
37   REAL *Lap, *L;
38 };
39
40 struct node {
41   REAL ctime;
42   REAL energy;
43    struct node* next;
44 };
45
46 int i;
47
48 //
49 // This series of kernels compartmentalizes the calculation into
      peices that do not depend serially on data.
50 // The kernels are called int the main function below.
51 //
52
53 //
54 // kernE1 and kernE2 are called in the FCHenergy function that
      calculates the FCH energy of the solution
55 //
56
57 __global__ void kernE1(int n, COMPLEX *U, REAL *Lap, REAL *Delx,
      REAL *Dely, REAL *Delz, COMPLEX *LapU, COMPLEX *DelxU, COMPLEX
      *DelyU, COMPLEX *DelzU)
58 {
59   int idx = blockIdx.x*blockDim.x+threadIdx.x;
60   if(idx < n)
61   {
62     U[idx].x = U[idx].x/n;
63     U[idx].y = U[idx].y/n;
64
65     LapU[idx].x = Lap[idx]*U[idx].x;
66     LapU[idx].y = Lap[idx]*U[idx].y;
67
68     DelxU[idx].x = -Delx[idx]*U[idx].y;
69     DelxU[idx].y = Delx[idx]*U[idx].x;
70
71     DelyU[idx].x = -Dely[idx]*U[idx].y;
```

```
72      DelyU[idx].y = Dely[idx]*U[idx].x;
73
74      DelzU[idx].x = -Delz[idx]*U[idx].y;
75      DelzU[idx].y = Delz[idx]*U[idx].x;
76
77    }
78  }
79
80  __global__ void kernE2(int n, REAL epsilon, REAL eta1, REAL eta2,
      REAL tau, REAL *Energy, COMPLEX *U, COMPLEX *LapU, COMPLEX *
      DelxU, COMPLEX *DelyU, COMPLEX *DelzU)
81  {
82    int idx = blockIdx.x*blockDim.x+threadIdx.x;
83    if(idx < n)
84    {
85      Energy[idx] = 0.5*(epsilon*epsilon*LapU[idx].x-(U[idx].x*U[idx
      ].x-1)*(U[idx].x+tau/2))*(epsilon*epsilon*LapU[idx].x-(U[idx].x
      *U[idx].x-1)*(U[idx].x+tau/2))-epsilon*(eta1*epsilon*epsilon
      /2*(DelxU[idx].x*DelxU[idx].x + DelyU[idx].x*DelyU[idx].x +
      DelzU[idx].x*DelzU[idx].x) + eta2*0.5*(U[idx].x+1)*(U[idx].x+1)
      *(0.5*(U[idx].x-1)*(U[idx].x-1)+tau/3*(U[idx].x-2)));
86    }
87  }
88
89  //
90  // kernLapL builds the Laplacian and L arrays to be used
      throughout the calculation
91  //
92
93  __global__ void kernLapL(int n, REAL epsilon, REAL eta1, REAL mu,
      REAL *Lap, REAL *L, REAL *Delx, REAL *Dely, REAL *Delz)
94  {
95    int idx = blockIdx.x*blockDim.x+threadIdx.x;
96    if(idx < n)
97    {
98      Lap[idx] = -(Delx[idx]*Delx[idx]+Dely[idx]*Dely[idx]+Delz[idx
      ]*Delz[idx]);
99      L[idx] = Lap[idx]*((epsilon*epsilon*Lap[idx] - mu + epsilon*
      eta1)*(epsilon*epsilon*Lap[idx] - mu));
100   }
101 }
102
103 //
104 // kernS1, kernS2, kernS3, and kernS4 accelerate peices of the
      function sub
```

```
105 //
106
107 __global__ void kernS1(int n, REAL tau, COMPLEX *V, COMPLEX *Q1)
108 {
109   int idx = blockIdx.x*blockDim.x+threadIdx.x;
110   if(idx < n)
111   {
112     Q1[idx].x = (0.5*(tau - 6) + V[idx].x)*V[idx].x*V[idx].x;
113     Q1[idx].y = (0.5*(tau - 6) + V[idx].y)*V[idx].y*V[idx].y;
114   }
115 }
116
117 __global__ void kernS2(int n, REAL epsilon, REAL mu, REAL *Lap,
    COMPLEX *V, COMPLEX *Q1, COMPLEX *R)
118 {
119   int idx = blockIdx.x*blockDim.x+threadIdx.x;
120   if(idx < n)
121   {
122     V[idx].x = V[idx].x/n;
123     V[idx].y = V[idx].y/n;
124
125     R[idx].x = (epsilon*epsilon*Lap[idx] - mu)*V[idx].x - Q1[idx].
    x/n;
126     R[idx].y = (epsilon*epsilon*Lap[idx] - mu)*V[idx].y - Q1[idx].
    y/n;
127   }
128 }
129
130 __global__ void kernS3(int n, REAL tau, COMPLEX *V, COMPLEX *R)
131 {
132   int idx = blockIdx.x*blockDim.x+threadIdx.x;
133   if(idx < n)
134   {
135     R[idx].x = (6 - tau - 3*V[idx].x)*V[idx].x*R[idx].x;
136     R[idx].y = (6 - tau - 3*V[idx].y)*V[idx].y*R[idx].y;
137   }
138 }
139
140 __global__ void kernS4(int n, REAL epsilon, REAL eta1, REAL eta2,
    REAL mu, REAL *Lap, COMPLEX *V, COMPLEX *Q1, COMPLEX *R)
141 {
142   int idx = blockIdx.x*blockDim.x+threadIdx.x;
143   if(idx < n)
144   {
```

```
145     R[idx].x = Lap[idx]*(-(epsilon*epsilon*Lap[idx] - mu + epsilon
        *eta1)*Q1[idx].x + R[idx].x + epsilon*(eta1 - eta2)*(mu*V[idx].
        x + Q1[idx].x));
146     R[idx].y = Lap[idx]*(-(epsilon*epsilon*Lap[idx] - mu + epsilon
        *eta1)*Q1[idx].y + R[idx].y + epsilon*(eta1 - eta2)*(mu*V[idx].
        y + Q1[idx].y));
147   }
148 }
149
150 //
151 // kern0, kern1, and kern2 accelerate portions of the main
        function.
152 //
153
154 __global__ void kern0(int n, COMPLEX *U, COMPLEX *V, int direct)
155 {
156   int idx = blockIdx.x*blockDim.x+threadIdx.x;
157   if(idx < n)
158   {
159     U[idx].x = V[idx].x - direct;
160     U[idx].y = 0.0;
161   }
162 }
163
164 __global__ void kern1(int n, REAL dt, REAL *L, COMPLEX *A, COMPLEX
        *V, COMPLEX *R)
165 {
166   int idx = blockIdx.x*blockDim.x+threadIdx.x;
167   if(idx < n)
168   {
169     A[idx].x = (exp(dt*L[idx])*V[idx].x + (exp(dt*L[idx]) - 1)/L[
        idx]*R[idx].x)/n;
170     A[idx].y = (exp(dt*L[idx])*V[idx].y + (exp(dt*L[idx]) - 1)/L[
        idx]*R[idx].y)/n;
171   }
172 }
173
174 __global__ void kern2(int n, REAL dt, REAL *L, COMPLEX *A, COMPLEX
        *V, COMPLEX *R, COMPLEX *R2)
175 {
176   int idx = blockIdx.x*blockDim.x+threadIdx.x;
177   if(idx < n)
178   {
179     A[idx].x = A[idx].x/n;
180     A[idx].y = A[idx].y/n;
```

```
181
182     V[idx].x = A[idx].x + (exp(dt*L[idx]) - 1 - dt*L[idx])/(dt*L[
        idx]*L[idx]*n)*(R2[idx].x - R[idx].x);
183     V[idx].y = A[idx].y + (exp(dt*L[idx]) - 1 - dt*L[idx])/(dt*L[
        idx]*L[idx]*n)*(R2[idx].y - R[idx].y);
184   }
185 }
186
187 //
188 // kernError computes the difference between stages that is used
       in the adaptive time stepping
189 //
190
191 __global__ void kernError(int n, COMPLEX *R, COMPLEX *V, COMPLEX *
       A)
192 {
193   int idx = blockIdx.x*blockDim.x+threadIdx.x;
194   if(idx < n)
195   {
196     R[idx].x = fabs(V[idx].x - A[idx].x);
197   }
198 }
199
200 //
201 // warpReduce, reduce, warpReduceSum, and reduceSum parallelize
       the reductions necessary to calculate energy and error
202 //
203
204 __device__ void warpReduce(volatile REAL *sdata, unsigned int tid,
        int blockSize)
205 {
206   if (blockSize >= 64) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 32]));
207   if (blockSize >= 32) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 16]));
208   if (blockSize >= 16) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 8]));
209   if (blockSize >= 8) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 4]));
210   if (blockSize >= 4) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 2]));
211   if (blockSize >= 2) sdata[tid] = max(fabs(sdata[tid]), fabs(
       sdata[tid + 1]));
212 }
213
```

```
214 __global__ void reduce(COMPLEX *g_idata, COMPLEX *g_odata, int n,
    int blockSize)
215 {
216   extern __shared__ REAL sdata[];
217   unsigned int tid = threadIdx.x;
218   unsigned int i = blockIdx.x*(blockSize*2) + tid;
219   unsigned int gridSize = blockSize*2*gridDim.x;
220   sdata[tid] = 0;
221
222   while (i < n) {sdata[tid] = max(fabs(g_idata[i].x),fabs(g_idata[
    i+blockSize].x)); i += gridSize; }
223   __syncthreads();
224
225   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = max(fabs(
    sdata[tid]), fabs(sdata[tid + 256])); } __syncthreads(); }
226   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = max(fabs(
    sdata[tid]), fabs(sdata[tid + 128])); } __syncthreads(); }
227   if (blockSize >= 128) { if (tid < 64) { sdata[tid] = max(fabs(
    sdata[tid]), fabs(sdata[tid + 64])); } __syncthreads(); }
228   if (tid < 32) warpReduce(sdata, tid, blockSize);
229   if (tid == 0) g_odata[blockIdx.x].x = sdata[0];
230 }
231
232 __device__ void warpReduceSum(volatile REAL *sdata, unsigned int
    tid, int blockSize)
233 {
234   if (blockSize >= 64) sdata[tid] = sdata[tid] + sdata[tid + 32];
235   if (blockSize >= 32) sdata[tid] = sdata[tid] + sdata[tid + 16];
236   if (blockSize >= 16) sdata[tid] = sdata[tid] + sdata[tid + 8];
237   if (blockSize >= 8) sdata[tid] = sdata[tid] + sdata[tid + 4];
238   if (blockSize >= 4) sdata[tid] = sdata[tid] + sdata[tid + 2];
239   if (blockSize >= 2) sdata[tid] = sdata[tid] + sdata[tid + 1];
240 }
241
242 __global__ void reduceSum(REAL *g_idata, REAL *g_odata, int n, int
     blockSize)
243 {
244   extern __shared__ REAL sdata[];
245   unsigned int tid = threadIdx.x;
246   unsigned int i = blockIdx.x*(blockSize*2) + tid;
247   unsigned int gridSize = blockSize*2*gridDim.x;
248   sdata[tid] = 0;
249
250   while (i < n) {sdata[tid] = g_idata[i] + g_idata[i+blockSize]; i
     += gridSize; }
```

```
251     __syncthreads();
252
253     if (blockSize >= 512) { if (tid < 256) { sdata[tid] = sdata[tid]
          + sdata[tid + 256]; } __syncthreads(); }
254     if (blockSize >= 256) { if (tid < 128) { sdata[tid] = sdata[tid]
          + sdata[tid + 128]; } __syncthreads(); }
255     if (blockSize >= 128) { if (tid < 64) { sdata[tid] = sdata[tid]
         + sdata[tid + 64]; } __syncthreads(); }
256     if (tid < 32) warpReduceSum(sdata, tid, blockSize);
257     if (tid == 0) g_odata[blockIdx.x] = sdata[0];
258 }
259
260 //
261 // FCHenergy computes the energy of the solution U at each
        timestep
262 //
263
264 REAL FCHenergy(dataBin &data, operBin oper, COMPLEX *U, REAL *
        Energy, REAL *Delx, REAL *Dely, REAL *Delz, COMPLEX *LapU,
        COMPLEX *DelxU, COMPLEX *DelyU, COMPLEX *DelzU, cufftHandle
        plan) {
265
266     EXEC (plan, U, U, CUFFT_FORWARD);
267
268     kernE1<<< data.dimGrid, data.dimBlock>>>(data.n, U, oper.Lap,
         Delx, Dely, Delz, LapU, DelxU, DelyU, DelzU);
269
270     EXEC(plan, LapU, LapU, CUFFT_INVERSE);
271     EXEC(plan, DelxU, DelxU, CUFFT_INVERSE);
272     EXEC(plan, DelyU, DelyU, CUFFT_INVERSE);
273     EXEC(plan, DelzU, DelzU, CUFFT_INVERSE);
274     EXEC(plan, U, U, CUFFT_INVERSE);
275
276     kernE2<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
         data.eta1, data.eta2, data.tau, Energy, U, LapU, DelxU, DelyU,
         DelzU);
277
278     reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
         REAL)>>>(Energy, Energy, data.n, data.blocksize);
279
280     int nRed = data.n/(data.blocksize*2);
281     while (nRed > 1)
282     {
283       reduceSum<<< data.dimGrid, data.dimBlock, data.blocksize*
         sizeof(REAL)>>>(Energy, Energy, nRed, data.blocksize);
```

```
284       nRed = nRed/(data.blocksize*2);
285     }
286
287     cudaMemcpy(data.Energy_h, Energy, sizeof(REAL),
          cudaMemcpyDeviceToHost);
288
289     return(*data.Energy_h);
290 }
291
292 //
293 // sub is used to compute both stages of the ETD-RK2 scheme
294 //
295
296 void sub(dataBin &data, operBin oper, COMPLEX *V, COMPLEX *Q1,
       COMPLEX *R, cufftHandle plan) {
297
298     kernS1<<< data.dimGrid, data.dimBlock>>>(data.n, data.tau, V, Q1
          );
299
300     EXEC(plan, V, V, CUFFT_FORWARD);
301     EXEC(plan, Q1, Q1, CUFFT_FORWARD);
302
303     cudaMemcpy(data.zf, V, sizeof(COMPLEX), cudaMemcpyDeviceToHost);
304
305     data.zf->x = data.zf->x / data.n;
306     data.zf->y = data.zf->y / data.n;
307
308     kernS2<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
          data.mu, oper.Lap, V, Q1, R);
309
310     EXEC(plan, V, V, CUFFT_INVERSE);
311     EXEC(plan, R, R, CUFFT_INVERSE);
312
313     kernS3<<< data.dimGrid, data.dimBlock>>>(data.n, data.tau, V, R)
          ;
314
315     EXEC(plan, V, V, CUFFT_FORWARD);
316     EXEC(plan, R, R, CUFFT_FORWARD);
317
318     kernS4<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
          data.eta1, data.eta2, data.mu, oper.Lap, V, Q1, R);
319
320 }
321
322 //
```

```
323 // addNode adds a node to the linked list that collects the energy
        and time step sizes
324 //
325
326 void addNode(struct node*& tail, REAL ctime) {
327   struct node* newNode = new node;
328   newNode->ctime = ctime;
329   newNode->energy = 0.0;
330   newNode->next = NULL;
331   tail->next = newNode;
332   tail = newNode;
333   }
334
335 //
336 // This is where the main function begins. To make sense of the
       code, begin here.
337 //
338
339 int main (int argc, char * const argv[]) {
340   if (argc <=1 )
341   {
342     cout << "Usage: " << argv[0] << " <filename> <output off/
       increasing/even (0/1/2)>" << endl;
343     exit(1);
344   }
345
346   time_t tic, toc;
347   time_t tic2, toc2;
348   time(&tic);
349   //
350   // Initialize necessary constants
351   //
352   cout << scientific;
353   cout.precision(10);
354   int i, j, k;
355
356   fstream myfile, myfile2, myfile3, myfile4;
357   stringstream sstm;
358   char datain[50];
359
360   REAL ctime, dtmax, Tmax, Tpicstep, TpicFactor, Lx, Ly, Lz, Ttol,
        Tfactor, Error;
361   int Nx, Ny, Nz, Mx, My, Mz;
362
363   REAL tPic = 1.0;
```

```
364   struct node* tEhead = NULL;
365   struct node* tEtail = NULL;
366
367   tEhead = new node;
368   tEhead->ctime = 0.0;
369   tEhead->energy = 0.0;
370   tEtail = tEhead;
371
372   string filein, fileUout, filetEout;
373
374   dataBin data;
375   operBin oper;
376
377   //
378   // Read in all the necessary data from the file given by the
       user
379   //
380
381   myfile3.open(argv[1], ios::in);
382
383   if(myfile3.is_open())
384   {
385     myfile3.ignore(512,'=');
386     myfile3 >> datain;
387     filein = datain;
388     cout << "Init cond file = " << filein << endl;
389
390     myfile3.ignore(512,'=');
391     myfile3 >> datain;
392     fileUout = datain;
393     cout << "Write out solution file = " << fileUout << ".dat" <<
       endl;
394
395     myfile3.ignore(512,'=');
396     myfile3 >> datain;
397     filetEout = datain;
398     cout << "Write out time and energy file = " << filetEout << ".
       dat" << endl;
399
400     myfile3.ignore(512,'=');
401     myfile3 >> datain;
402     data.epsilon = atof(datain);
403     cout << "epsilon = " << data.epsilon << endl;
404
405     myfile3.ignore(512,'=');
```

```
406     myfile3 >> datain;
407     data.tau = atof(datain);
408     data.mu = 2 - data.tau;
409     cout << "tau = " << data.tau << endl;
410
411     myfile3.ignore(512,'=');
412     myfile3 >> datain;
413     data.eta1 = atof(datain);
414     cout << "eta1 = " << data.eta1 << endl;
415
416     myfile3.ignore(512,'=');
417     myfile3 >> datain;
418     data.eta2 = atof(datain);
419     cout << "eta2 = " << data.eta2 << endl;
420
421     myfile3.ignore(512,'=');
422     myfile3 >> datain;
423     data.dt = atof(datain);
424     cout << "dt = " << data.dt << endl;
425
426     myfile3.ignore(512,'=');
427     myfile3 >> datain;
428     dtmax = atof(datain);
429     cout << "dtmax = " << dtmax << endl;
430
431     myfile3.ignore(512,'=');
432     myfile3 >> datain;
433     Tmax = atof(datain);
434     cout << "Tmax = " << Tmax << endl;
435
436     myfile3.ignore(512,'=');
437     myfile3 >> datain;
438     Tpicstep = atof(datain);
439     cout << "Tpicstep = " << Tpicstep << endl;
440
441     myfile3.ignore(512,'=');
442     myfile3 >> datain;
443     TpicFactor = atof(datain);
444     cout << "TpicFactor = " << TpicFactor << endl;
445
446     myfile3.ignore(512,'=');
447     myfile3 >> datain;
448     Ttol = atof(datain);
449     cout << "Ttol = " << Ttol << endl;
450
```

```
451     myfile3.ignore(512,'=');
452     myfile3 >> datain;
453     Tfactor = atof(datain);
454     cout << "Tfactor = " << Tfactor << endl;
455 
456     myfile3.ignore(512,'=');
457     myfile3 >> datain;
458     Nx = atoi(datain);
459     cout << "Nx = " << Nx << endl;
460 
461     myfile3.ignore(512,'=');
462     myfile3 >> datain;
463     Ny = atoi(datain);
464     cout << "Ny = " << Ny << endl;
465 
466     myfile3.ignore(512,'=');
467     myfile3 >> datain;
468     Nz = atoi(datain);
469     cout << "Nz = " << Nz << endl;
470 
471     myfile3.ignore(512,'=');
472     myfile3 >> datain;
473     Lx = atof(datain);
474     cout << "Lx = " << Lx << endl;
475 
476     myfile3.ignore(512,'=');
477     myfile3 >> datain;
478     Ly = atof(datain);
479     cout << "Ly = " << Ly << endl;
480 
481     myfile3.ignore(512,'=');
482     myfile3 >> datain;
483     Lz = atof(datain);
484     cout << "Lz = " << Lz << endl;
485 
486     myfile3.ignore(512,'=');
487     myfile3 >> datain;
488     data.blocksize = atoi(datain);
489     cout << "Block size = " << data.blocksize << endl;
490     cout << "Grid size = " << Nx*Ny*Nz/data.blocksize << endl;
491   }
492   else { cout << "File could not be read." << endl; }
493 
494   ctime = 0.0;
495 
```

```
496   tPic = TpicFactor;
497
498   if (*argv[2] == '2')
499   {
500     tPic = Tpicstep;
501   }
502
503   Mx = Nx/2;
504   My = Ny/2;
505   Mz = Nz/2;
506   data.n = Nx*Ny*Nz;
507   data.mu = 2 - data.tau;
508   data.dimBlock = data.blocksize;
509   data.dimGrid = data.n/data.dimBlock.x;
510
511
512   //
513   // Build the laplacian array, and dynamically allocate other
       needed arrays
514   //
515
516   REAL *xkvec, *ykvec, *zkvec;
517   xkvec = new REAL[Nx];
518   ykvec = new REAL[Ny];
519   zkvec = new REAL[Nz];
520
521   for (i=0; i<Mx; i++)
522   {
523     xkvec[i] = i*2*PI/Lx;
524     xkvec[i+Mx] = (i-Mx)*2*PI/Lx;
525   }
526
527   for (i=0; i<My; i++)
528   {
529     ykvec[i] = i*2*PI/Ly;
530     ykvec[i+My] = (i-My)*2*PI/Ly;
531   }
532
533   for (i=0; i<Mz; i++)
534   {
535     zkvec[i] = i*2*PI/Lz;
536     zkvec[i+Mz] = (i-Mz)*2*PI/Lx;
537   }
538
539   REAL *Delx_h, *Dely_h, *Delz_h, *Delx, *Dely, *Delz, *Energy;
```

```
540   data.Energy_h = new REAL;
541   Delx_h = new REAL[data.n];
542   Dely_h = new REAL[data.n];
543   Delz_h = new REAL[data.n];
544
545   for (i=0; i<Nx; i++) {
546     for (j=0; j<Ny; j++) {
547       for (k=0; k<Nz; k++)
548       {
549         Delx_h[k+Nz*(j+Ny*i)] = xkvec[i];
550         Dely_h[k+Nz*(j+Ny*i)] = ykvec[j];
551         Delz_h[k+Nz*(j+Ny*i)] = zkvec[k];
552       }
553     }
554   }
555
556   cout << "Del operators built" << endl;
557
558   //
559   // Initialize all of the arrays that will be used in fourier
       transforms, and build the plans necessary
560   //
561
562   int nBytes = sizeof(COMPLEX)*data.n;
563   cout << "nBytes is = " << nBytes << endl;
564  COMPLEX *U_h, *U, *V, *A, *Q1, *R, *R2, *LapU, *max;
565
566   //
567   // Read in inital U data from filein
568   //
569
570   U_h = new COMPLEX[nBytes];
571   data.max_h = new COMPLEX;
572   data.zf = new COMPLEX;
573
574   myfile.open(filein.c_str(),ios::in);
575
576   if(myfile.is_open())
577   {
578     for (i=0; i<data.n; i++)
579     {
580       myfile.getline(datain, 50);
581       U_h[i].x = atof(datain);
582       U_h[i].y = 0.0;
583     }
```

```
584     cout << "Initial data read in successfully" << endl;
585   } else { cout << "Initial data not read" << endl; }
586
587   myfile.close();
588
589   cudaMalloc((void **)&U, nBytes);
590   cudaMalloc((void **)&V, nBytes);
591   cudaMalloc((void **)&A, nBytes);
592   cudaMalloc((void **)&Q1, nBytes);
593   cudaMalloc((void **)&R, nBytes);
594   cudaMalloc((void **)&R2, nBytes);
595   cudaMalloc((void **)&LapU, nBytes);
596   cudaMalloc((void **)&Energy, sizeof(REAL)*data.n);
597
598   cudaMalloc((void **)&oper.Lap, sizeof(REAL)*data.n);
599   cudaMalloc((void **)&oper.L, sizeof(REAL)*data.n);
600   cudaMalloc((void **)&Delx, sizeof(REAL)*data.n);
601   cudaMalloc((void **)&Dely, sizeof(REAL)*data.n);
602   cudaMalloc((void **)&Delz, sizeof(REAL)*data.n);
603   cudaMalloc((void **)&max, sizeof(COMPLEX)*data.dimGrid.x/2);
604
605   cufftHandle plan;
606   cufftPlan3d(&plan, Nx, Ny, Nz, TYPE);
607
608
609   cudaMemcpy(Delx, Delx_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
610   cudaMemcpy(Dely, Dely_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
611   cudaMemcpy(Delz, Delz_h, sizeof(REAL)*data.n,
       cudaMemcpyHostToDevice);
612   cudaMemcpy(U, U_h, nBytes, cudaMemcpyHostToDevice);
613   cout << "Initial data copied" << endl;
614
615   //
616   // Calculate the constant arrays Lap and L that will be used
       below
617   //
618   kernLapL<<< data.dimGrid, data.dimBlock>>>(data.n, data.epsilon,
        data.eta1, data.mu, oper.Lap, oper.L, Delx, Dely, Delz);
619   cout << "Lap and L initialized" << endl;
620
621   sstm << filetEout << ".dat";
622   myfile4.open(sstm.str().c_str(),ios::out);
623   myfile4.precision(15);
```

```
624
625   tEtail->energy = FCHenergy(data, oper, U, Energy, Delx, Dely,
       Delz, LapU, Q1, R, R2, plan);
626   myfile4 << ctime << "\t" << tEhead->energy << endl;
627   cout << time << "\t" << 0.0 << "\t" << tEtail->energy << "\t" <<
       (REAL) 0.0 << endl;
628
629   kern0<<< data.dimGrid, data.dimBlock>>>(data.n, V, U, (REAL) -1)
       ;
630
631   //
632   // while loop governing time stepping
633   //
634   while (ctime < Tmax)
635   {
636     time(&tic2);
637
638     //
639     // Calculate A and V
640     //
641
642     sub(data, oper, V, Q1, R, plan);
643
644     kern1<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, oper.
      L, A, V, R);
645
646     cudaMemcpy(A, data.zf, sizeof(COMPLEX), cudaMemcpyHostToDevice
      );
647
648     EXEC(plan, A, A, CUFFT_INVERSE);
649
650     sub(data, oper, A, Q1, R2, plan);
651
652     kern2<<< data.dimGrid, data.dimBlock>>>(data.n, data.dt, oper.
      L, A, V, R, R2);
653
654     cudaMemcpy(V, data.zf, sizeof(COMPLEX), cudaMemcpyHostToDevice
      );
655
656     EXEC(plan, V, V, CUFFT_INVERSE);
657     EXEC(plan, A, A, CUFFT_INVERSE);
658
659     kernError<<< data.dimGrid, data.dimBlock>>>(data.n, R, V, A);
660
```

```
661     reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof(
    REAL)>>>(R, max, data.n, data.blocksize);
662     int nRed = data.n/(data.blocksize*2);
663     while (nRed > 1)
664     {
665       reduce<<< data.dimGrid, data.dimBlock, data.blocksize*sizeof
    (REAL)>>>(max, max, nRed, data.blocksize);
666       nRed = nRed/(data.blocksize*2);
667     }
668
669     cudaMemcpy(data.max_h, max, sizeof(REAL),
    cudaMemcpyDeviceToHost);
670
671     Error = data.max_h->x;
672
673     //
674     // Update time and update U to be the solution at the next
    time step
675     //
676
677     ctime = ctime + data.dt;
678
679     kern0<<< data.dimGrid, data.dimBlock>>>(data.n, U, V, 1);
680
681     tEtail->energy = FCHenergy(data, oper, U, Energy, Delx, Dely,
    Delz, LapU, Q1, R, R2, plan);
682
683     time(&toc2);
684
685     myfile4 << ctime << "\t" << tEtail->energy << endl;
686     cout << ctime << "\t" << data.dt << "\t" << tEtail->energy <<
    "\t" << difftime(toc2,tic2) << endl;
687
688     //
689     // Calculate the adaptive time step
690     //
691
692     data.dt = min(data.dt*min(Tfactor*sqrt(Ttol/Error), (REAL)
    1.3), dtmax);
693
694     if (abs(ctime - tPic) < 0.0000000001 && *argv[2] != '0')
695     {
696       cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
697
```

```
698       sstm << fileUout << (int) floor(tPic/TpicFactor + 0.5) << ".
    dat";
699       myfile2.open(sstm.str().c_str(),ios::out);
700       if (myfile2.is_open())
701       {
702
703         myfile2.precision(15);
704         for (i=0; i<data.n; i++)
705         {
706           myfile2 << U_h[i].x << endl;
707         }
708
709       }
710       cout << "Writing out " << sstm.str() << endl;
711       myfile2.close();
712       sstm.str("");
713
714       if (*argv[2] == '1')
715       {
716         tPic += pow((REAL) 10.0,(REAL) ceil(log10(tPic+TpicFactor)
    )-1);
717       }
718       else
719       {
720         tPic += Tpicstep;
721       }
722     }
723     //
724     // Adjust the timestep if necessary
725     //
726     if (data.dt > (Tmax - ctime))
727     {
728       data.dt = Tmax - ctime;
729     }
730
731     if (data.dt > (tPic - ctime) && *argv[2] != '0')
732     {
733       data.dt = tPic - ctime;
734     }
735
736     addNode(tEtail, ctime);
737
738   }
739
```

```
740   tEtail->energy = FCHenergy(data, oper, U, Energy, Delx, Dely,
      Delz, LapU, Q1, R, R2, plan);
741
742   //
743   // Write out final solution data
744   //
745
746   sstm << fileUout << (int) floor(Tmax/TpicFactor + 0.5) << ".dat"
      ;
747   myfile2.open(sstm.str().c_str(),ios::out);
748   if (myfile2.is_open())
749   {
750     cudaMemcpy(U_h, U, nBytes, cudaMemcpyDeviceToHost);
751
752     myfile2.precision(15);
753     for (i=0; i<data.n; i++)
754     {
755       myfile2 << U_h[i].x << endl;
756     }
757
758   }
759   cout << "Writing out " << sstm.str() << endl;
760   myfile2.close();
761   sstm.str("");
762
763   while (tEhead != NULL)
764   {
765     tEtail = tEhead;
766     tEhead = tEhead->next;
767     delete tEtail;     }
768
769   myfile4 << ctime << "\t" << tEhead->energy << endl;
770   myfile4.close();
771   delete tEhead;
772
773   //
774   // Clean up all the memory used on the GPU and CPU
775   //
776
777   cudaFree(U);
778   cudaFree(U_h);
779   cudaFree(V);
780   cudaFree(A);
781   cudaFree(Q1);
782   cudaFree(R);
```

```
783   cudaFree(R2);
784   cudaFree(LapU);
785   cudaFree(oper.Lap);
786   cudaFree(oper.L);
787   cudaFree(max);
788
789   cufftDestroy(plan);
790
791   delete data.Energy_h;
792   delete data.max_h;
793   delete[] U_h;
794   delete[] xkvec;
795   delete[] ykvec;
796   delete[] zkvec;
797   delete[] Delx_h;
798   delete[] Dely_h;
799   delete[] Delz_h;
800
801   time(&toc);
802   cout << "\n" << difftime(toc,tic) << endl;
803
804     return 0;
805 }
```

./code/FCHExInt.cu

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] L. D. Landau. On the theory of phase transitions, part ii. *Sov. Phys. JETP*, 11:627, 1937.

[2] L. D. Landau. On the theory of phase transitions, part i. *Sov. Phys. JETP*, 7:19ff, 1937.

[3] V. L. Ginzburg and L. D. Landau. Concerning the theory of superconductivity. *Soviet Physics IETP*, 20:1064–1082, 1950.

[4] G. J. Fix. Phase field models for free boundary problems. *Free boundary problems: Theory and applications*, 2:580–589, 1983.

[5] J. S. Langer. Models of pattern formation in first-order phase transitions. *Dir. in Cond. Matter Phys., World Sci. Pub.*, pages 164–186, 1986.

[6] L. Q. Chen. Phase-field models for microstructure evolution. *Annu. Rev. Mat. Res.*, 32(1):113–140, 2002.

[7] S. Wise, J. Kim, and J. Lowengrub. Solving the regularized, strongly anisotropic cahn-hilliard equation by an adaptive nonlinear multigrid method. *J. Comp. Phys.*, 226(1):414–446, 2007.

[8] J. S. Rowlinson. Translation of jd van der waals'the thermodynamik theory of capillarity under the hypothesis of a continuous variation of density. *J. Stat. Phys.*, 20(2):197–200, 1979.

[9] J. D. Van der Waals. The thermodynamic theory of capillarity under the hypothesis of a continuous variation of density. *J. Stat. Phys.*, 20(2):200–244, 1979.

[10] J. W. Cahn and J. E. Hilliard. Free energy of a nonuniform system. i. interfacial free energy. *J. Chem. Phys.*, 28:258, 1958.

[11] R. L. Pego. Front migration in the nonlinear cahn-hilliard equation. *Proc. R. Soc. A: Math. Phys. and Eng. Sci.*, 422(1863):261, 1989.

[12] J. Philibert. One and a half century of diffusion: Fick, einstein, before and beyond. *Diffusion Fundamentals*, 2(1):1–10, 2005.

[13] C. M. Elliott and Z. Songmu. On the cahn-hilliard equation. *Arch. Ration. Mech. Anal.*, 96(4):339–357, 1986.

[14] N. D. Alikakos, P. W. Bates, and X. Chen. Convergence of the cahn-hilliard equation to the hele-shaw model. *Arch. Ration. Mech. Anal.*, 128(2):165–205, 1994.

[15] L. Modica. The gradient theory of phase transitions and the minimal interface criterion. *Arch. Ration. Mech. Anal.*, 98(2):123–142, 1987.

[16] P. Sternberg. The effect of a singular perturbation on nonconvex variational problems. *Arch. Ration. Mech. Anal.*, 101(3):209–260, 1988.

[17] P. De Mottoni and M. Schatzman. Geometrical evolution of developed interfaces. *Amer. Math. Soc.*, 347(5), 1995.

[18] P. B. Canham. The minimum energy of bending as a possible explanation of the biconcave shape of the human red blood cell*. *J. Theor. Bio.*, 26(1):61–81, 1970.

[19] W. Helfrich. Elastic properties of lipid bilayers: theory and possible experiments. *Z. Naturforsch*, 28(11):693–703, 1973.

[20] Q. Du, C. Liu, and X. Wang. Simulating the deformation of vesicle membranes under elastic bending energy in three dimensions. *J. Comp. Phys.*, 212(2):757–777, 2006.

[21] R. D. Kamien. The geometry of soft materials: a primer. *Rev. Mod. Phys.*, 74:953, 2002.

[22] M. E. Gurtin and M. E. Jabbour. Interface evolution in three dimensions with curvature-dependent energy and surface diffusion: Interface-controlled evolution, phase transitions, epitaxial growth of elastic films. *Arch. Ration. Mech. Anal.*, 163(3):171–208, 2002.

[23] K. Promislow and B. Wetton. Pem fuel cells: a mathematical overview. *SIAM J. Appl. Math*, 70(2):369–409, 2009.

[24] S. J. Paddison. *Device and materials modeling in PEM fuel cells*, volume 113. Springer Verlag, 2009.

[25] S. Dai and K. Promislow. Geometric evolution of bi-layers under the functionalized cahn-hilliard equation. *Proc. R. Soc. A*, 2013.

[26] G. Gompper, M. Schick, C. Domb, and J. L. Lebowitz. *Phase Transitions and Critical Phenomena: Self-assembling Amphiphilic Systems*. Academic Press, 1994.

[27] G. Gompper and M. Schick. Correlation between structural and interfacial properties of amphiphilic systems. *Phys. Rev. Letters*, 65(9):1116–1119, 1990.

[28] N. Gavish, G. Hayrapetyan, K. Promislow Promislow, and L. Bronsard. Curvature driven flow of bi-layer interfaces. *Physica D: Nonlinear Phenomena*, 240(7):675 – 693, 2011.

[29] W. Y. Hsu and T. D. Gierke. Ion transport and clustering in nafion perfluorinated membranes. *J. Memb. Sci.*, 13(3):307–326, 1983.

[30] A. S. Ioselevich, A. A. Kornyshev, and J. H. G. Steinke. Fine morphology of proton-conducting ionomers. *J. Phys. Chem. B*, 108(32):11953–11963, 2004.

[31] L. Rubatat, G. Gebel, and O. Diat. Fibrillar structure of nafion: Matching fourier and real space studies of corresponding films and solutions. *Macromolecules*, 37(20):7772–7783, 2004.

[32] K. Schmidt-Rohr and Q. Chen. Parallel cylindrical water nanochannels in nafion fuel-cell membranes. *Nature Materials*, 7(1):75–83, 2007.

[33] W. G. F. Grot, G. E. Munn, and P. N. Walmsley. Perfluorinated ion exchange membranes. In *141th National Meeting, The Electrochemical Society, Houston, Texas*, 1972.

[34] D. J. Eyre. Unconditionally gradient stable time marching the cahn-hilliard equation. In *Computational and mathematical models of microstructural evolution*, volume 529 of *Materials Research Society Symposia Proceedings*, pages 39–64, Warrendale, PA, 1998.

[35] T. Kupper and N. Masbaum. Simulation of particle growth and ostwald ripening via the cahn-hilliard equation. *Acta metallurgica et materialia*, 42(6):1847–1858, 1994.

[36] C. M. Elliott and D. A. French. Numerical studies of the cahn-hilliard equation for phase separation. *IMA J. Appl. Math.*, 38(2):97, 1987.

[37] C. M. Elliott, D. A. French, and F. A. Milner. A second order splitting method for the cahn-hilliard equation. *Numerische Mathematik*, 54(5):575–590, 1989.

[38] C. M. Elliott and D. A. French. A nonconforming finite-element method for the two-dimensional cahn-hilliard equation. *SIAM J. Num. Anal.*, pages 884–903, 1989.

[39] Q. Du and R. A. Nicolaides. Numerical analysis of a continuum model of phase transition. *SIAM J. Num. Anal.*, 28(5):1310–1322, 1991.

[40] J. W. Barrett, J. F. Blowey, and H. Garcke. Finite element approximation of the cahn-hilliard equation with degenerate mobility. *SIAM J. Num. Anal.*, 37(1):286–318, 1999.

[41] X. Feng and A. Prohl. Error analysis of a mixed finite element method for the cahn-hilliard equation. *Numerische Mathematik*, 99(1):47–84, 2004.

[42] D. Kay and R. Welford. A multigrid finite element solver for the cahn-hilliard equation. *J. Comp. Phys.*, 212(1):288–304, 2006.

[43] J. Kim, K. Kang, and J. Lowengrub. Conservative multigrid methods for cahn-hilliard fluids. *J. Comp. Phys.*, 193(2):511–543, 2004.

[44] Z. Hu, S. M. Wise, C. Wang, and J. S. Lowengrub. Stable and efficient finite-difference nonlinear-multigrid schemes for the phase field crystal equation. *J. Comp. Phys.*, 228(15):5323–5339, 2009.

[45] S. M. Wise. Unconditionally stable finite difference, nonlinear multigrid simulation of the cahn-hilliard-hele-shaw system of equations. *J. Sci. Comp.*, 44(1):38–68, 2010.

[46] D.J. Eyre. An unconditionally stable one-step scheme for gradient systems. *Unpublished article*, 1998.

[47] D. Furihata. A stable and conservative finite difference scheme for the cahn-hilliard equation. *Numerische Mathematik*, 87(4):675–699, 2001.

[48] J. Zhu, L. Q. Chen, J. Shen, and V. Tikare. Coarsening kinetics from a variable-mobility cahn-hilliard equation: Application of a semi-implicit fourier spectral method. *Phys. Rev. E*, 60(4):3564, 1999.

[49] B. P. Vollmayr-Lee and A. D. Rutenberg. Fast and accurate coarsening simulation with an unconditionally stable time step. *Phys. Rev. E*, 68(6):066703, 2003.

[50] X. Ye and X. Cheng. The fourier spectral method for the cahn-hilliard equation. *Appl. Math. Comp.*, 171(1):345–357, 2005.

[51] J. Shen and X. Yang. Numerical approximations of allen-cahn and cahn-hilliard equations. *Discr. and Cont. Dynam. Sys.*, 28(4):1669–1691, 2010.

[52] G. N. Wells, E. Kuhl, and K. Garikipati. A discontinuous galerkin method for the cahn-hilliard equation. *J. Comp. Phys.*, 218(2):860–877, 2006.

[53] Y. Xia, Y. Xu, and C. W. Shu. Local discontinuous galerkin methods for the cahn-hilliard type equations. *J. Comp. Phys.*, 227(1):472–491, 2007.

[54] S. M. Wise, C. Wang, and J. S. Lowengrub. An energy-stable and convergent finite-difference scheme for the phase field crystal equation. *SIAM J. Num. Anal.*, 47:2269, 2009.

[55] H. Gomez and T. J. R. Hughes. Provably unconditionally stable, second-order time-accurate, mixed variational methods for phase-field models. *J. Comp. Phys.*, 230(13):5310–5327, 2011.

[56] J. Shen and X. Yang. Energy stable schemes for cahn-hilliard phase-field model of two-phase incompressible flows. *Chinese Annals of Mathematics, Series B*, 31(5):743–758, 2010.

[57] D. A. French. Computations on the cahn-hilliard model of solidification. *Appl. Math. Comp.*, 40(1):55–76, 1990.

[58] M. Kronbichler and G. Kreiss. A hybrid level-set-cahn-hilliard model for two-phase flow. In *1st European Conference on Microfluidics, Bologna, Italy*, 2008.

[59] Mark Ryerson Willoughby. *High-Order Time-Adaptive Numerical Methods For The Allen-Cahn and Cahn-Hilliard Equations*. PhD thesis, University of British Columbia, 2011.

[60] J Certaine. The solution of ordinary differential equations with large time constants. *Mathematical methods for digital computers*, pages 128–132, 1960.

[61] D. A. Pope. An exponential method of numerical integration of ordinary differential equations. *Comm. of the ACM*, 6(8):491–493, 1963.

[62] B. V. Minchev, W. M. Wright, et al. A review of exponential integrators for first order semi-linear problems. *Model. Ident. Cont.*, 27(4):201, 2006.

[63] T. Y. Hou, J. S. Lowengrub, and M. J. Shelley. Removing the stiffness from interfacial flows with surface tension. *J. Comp. Phys.*, 114(2):312–338, 1994.

[64] G. Beylkin, J. M. Keiser, and L. Vozovoi. A new class of time discretization schemes for the solution of nonlinear pdes. *J. Comp. Phys.*, 147(2):362–387, 1998.

[65] S. M. Cox and P. C. Matthews. Exponential time differencing for stiff systems. *J. Comp. Phys.*, 176(2):430–455, 2002.

[66] Q. Du and W. Zhu. Stability analysis and applications of the exponential time differencing schemes. *J. Comp. Math.*, 22(2):200–209, 2004.

[67] Q. Du and W. Zhu. Analysis and applications of the exponential time differencing schemes and their contour integration modifications. *BIT Num. Math.*, 45(2):307–328, 2005.

[68] A. Kassam. *High order timestepping for stiff semilinear partial differential equations.* PhD thesis, University of Oxford, 2004.

[69] A. Kassam and L. N. Trefethen. Fourth-order time-stepping for stiff pdes. *SIAM J. Sci. Comp.*, 26(4):1214–1233, 2005.

[70] M. Hochbruck and A. Ostermann. Exponential runge–kutta methods for parabolic problems. *Appl. Numer. Math.*, 53(2):323–339, 2005.

[71] M. Hochbruck and A. Ostermann. Explicit exponential runge–kutta methods for semilinear parabolic problems. *SIAM J. Num. Anal.*, 43(3):1069–1090, 2005.

[72] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numerica*, 19(1):209–286, 2010.

[73] M. Tokman. Efficient integration of large stiff systems of odes with exponential propagation iterative (epi) methods. *J. Comp. Phys.*, 213(2):748–776, 2006.

[74] M. Tokman and J. Loffeld. Efficient design of exponential-krylov integrators for large scale computing. *Procedia Computer Science*, 1(1):229–237, 2010.

[75] M. Tokman. A new class of exponential propagation iterative methods of runge–kutta type (epirk). *J. Comp. Phys.*, 230(24):8762–8778, 2011.

[76] A. Koskela and A. Ostermann. Exponential taylor methods: Analysis and implementation. *Comp. Math. Applic.*, 2012.

[77] L. Trefethen. *Spectral Methods in MATLAB*. Society for Indus- trial and Applied Mathematics, 2000.

[78] A. Christlieb and B. Ong. Implicit parallel time integrators. *J. Sci. Comp.*, pages 1–13, 2010.

[79] Z. Xu, A. Christlieb, and K. Promislow. On the unconditionally gradient stable scheme for the cahn-hilliard equation and its implementation with fourier method. *Comm. Math. Sci.*, 11(2):345, 2013. This paper will be submitted for publication within the next six months.

[80] Nvidia's next generation cuda compute architecture: Fermi. Technical report, Nvidia, http://www.nvidia.com/object/fermi_architecture.html, 2011.

[81] S. Jain and F. S. Bates. Consequences of nonergodicity in aqueous binary peo-pb micellar dispersions. *Macromolecules*, 37(4):1511–1523, 2004.

[82] D. Bendejacq, M. Joanicot, and V. Ponsinet. Pearling instabilities in water-dispersed copolymer cylinders with charged brushes. *Euro. Phys. J. E: Soft Matter and Bio. Phys.*, 17(1):83–92, 2005.

[83] L. G. Reyna and M. J. Ward. Metastable internal layer dynamics for the viscous cahn-hilliard equation. *Meth. Appl. Anal.*, 2:285–306, 1995.

[84] A. J. Bray. Theory of phase- ordering kinetics. *Adv. Phys.*, 43:357, 1994.

[85] P. Yue, J. Feng, C. Liu, and J. Shen. A diffuse-interface method for simulating two-phase flows of complex fluids. *J. Fluid. Mech.*, 515:293–317, 2004.

[86] D. M. Anderson, G. B. McFadden, and A. A. Wheeler. Diffuse-interface methods in fluid mechanics. *Annu. Rev. Fluid. Mech*, 30:139–165, 1998.

[87] S. M. Allen and J. W. Cahn. A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening. *Acta Metall. Mater.*, 27:1085–1095, 1979.

[88] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin Heidelberg, 1991.

[89] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I - Nonstiff problems*. Springer-Verlag Berlin Heidelberg, 2 edition, 1993.

[90] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, Pa, 2nd edition, 2003.

[91] A. Christlieb, K. Promislow, and Z. Xu. On the unconditionally gradient stable scheme for the cahn-hilliard equation and its implementation with fourier method. *Comm. Math. Sci*, 2013.

[92] N. Gavish, J. Jones, Z. Xu, A. Christlieb, and K. Promislow. Variational models of network formation and ion transport: applications to perfluorosulfonate ionomer membranes. *Polymers*, 4:630–655, 2012.

[93] L. Bronsard and F. Reitich. On three-phase boundary motion and the singular limit of a vector-valued ginzburg–landau equation. *Arch. Ration. Mech. Anal.*, 124(355–379), 1993.

[94] J. Carr and R. Pego. Metastable patterns in solutions of $u_t = \epsilon^2 u_{xx} - f(u)$. *Comm. Pure Appl. Math*, 42:523–576, 1989.

[95] L.C. Evans. *Partial Differential Equations*. American Mathematical Society, 2 edition, 2010.

[96] J. C. Gower. Properties of euclidean and non-euclidean distance matrices. *Lin. Alg. Applic.*, 67:81–97, 1985.

[97] R. Alexander. Diagonally implicit runge-kutta methods for stiff odes. *SIAM J. Num. Anal.*, 14:1006–1021, 1977.

[98] M. R. Willoughby. High-order time adaptive numerical methods for the allen-cahn and cahn-hilliard equations. Msc thesis, Institute of Applied Mathematics, University of British Columbia, 2011.

[99] J. J. B. de Swart and G. Söderland. On the construction of error estimators for implicit runge-kutta methods. *J. Comp. Appl. Math.*, 86:347–358, 1997.

[100] A. Dutt, L. Greengard, and V. Rokhlin. Spectral deferred correction methods for ordinary differential equations. *BIT*, 40:241–266, 2000.

[101] S. Boscarino. On an accurate third order implicit-explicit runge-kutta method for stiff problems. *Appl. Numer. Math*, 59:1515–1528, 2009.

[102] U. M. Ascher, S. J. Ruuth, and B. R. Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM J. Num. Anal.*, 32:797–823, 1995.

[103] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Appl. Numer. Math*, 25:151–167, 1997.

[104] M.L. Minion. Semi-implicit spectrally deferred correction methods for ordinary differential equations. *Comm. Math. Sci*, 1:471–500, 2003.

[105] M. Cheng and A. D. Rutenberg. Maximally fast coarsening algorithms. *Phys. Rev. E*, 72:055701(R), 2005.

[106] M. Cheng and J. A. Warren. Controlling the accuracy of unconditionally stable algorithms in the cahn-hilliard equation. *Phys. Rev. E*, 75:017702, 2007.

[107] C. Wang, X. Wang, and S. M. Wise. Unconditionally stable schemes for equations of thin film epitaxy. *Discrete Cont. Dyn. Sys. Ser. A*, 28:405–423, 2010.

[108] J. Shen, C. Wang, X. Wang, and S. M. Wise. Second-order convex splitting schemes for gradient flows with ehrlich-schwoebel-type energy: Application to thin film epitaxy. *SIAM J. Num. Anal.*, 50:105–125, 2012.

[109] A. Christlieb, J. Jones, K. Promislow, B. Wetton, and M. Willoughby. High accuracy solutions to energy gradient flows from material science models. *J. Comp. Phys.*, 2013.

[110] N. J. Higham. *Accuracy and Stability of Numberical Algorithms*. Number 48. Siam, 1996.

[111] W. Feng and K. W. Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[112] W. Shinoda, T. Nakamura, and S. O. Nielsen. Free energy analysis of vesicle-to-bicelle transformation. *Soft Matter*, 7(19):9012–9020, 2011.

[113] S. Yamamoto, Y. Maruyama, and S. Hyodo. Dissipative particle dynamics study of spontaneous vesicle formation of amphiphilic molecules. *J. Chem. Phys.*, 116:5842, 2002.