LONG TERM DYNAMICS OF THE DI-BLOCK COPOLYMER MODEL ON HIGHER DIMENSIONAL DOMAINS

by

Michael R. Atkins
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Mathematics

Committee:

_____	Thomas Wanner, Thesis Director
_____	Evelyn Sander, Committee Member
_____	Daniel M. Anderson, Committee Member
_____	Stephen H. Saperstone, Acting Chairman, Department of Mathematical Sciences
_____	Richard Diecchio, Associate Dean for Academic and Student Affairs, College of Science
_____	Vikas Chandhoke, Dean, College of Science

Date: 5/6/2011

Spring Semester 2011
George Mason University
Fairfax, VA

Long Term Dynamics of the Di-Block Copolymer Model on Higher Dimensional Domains

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Michael R. Atkins
Bachelor of Science
George Mason University, 2010

Director: Thomas Wanner, Professor
Department of Mathematical Sciences

Spring Semester 2011
George Mason University
Fairfax, VA

Dedication

This thesis is dedicated to the memory of Ann Atkins.

Acknowledgments

I would like to thank the following people who made this possible.

- Thomas Wanner
- Evelyn Sander
- Ian Johnson
- Jeff Snider
- Pearu Peterson
- The developers of matplotlib, Python and numpy
- Dept. of Mathematical Sciences
- Anyone who feels their name should be here instead of this sentence.

Table of Contents

Page

List of Tables vii
List of Figures viii
Abstract ix
1 Introduction 1
1.1 Model 1
1.2 Galerkin's Method 6
1.3 Fourier Analysis 8
1.3.1 A Basis of Eigenfunctions 8
1.3.2 A Fact About Fourier Coefficients 11
1.4 Semi Implicit Method 13
2 Numerical Scheme 18
2.1 Derivation 18
2.2 Implementation in C 20
2.2.1 Fast Fourier Transform 21
2.2.2 Zero Padding 22
2.3 Implementation in Python 23
3 Results and Conclusions 25
3.1 Creating a Double Gyroid 25
3.2 Examining a Phase Diagram 26
3.3 Conclusion 28
A Notation 33
B AISE 34
B.0.1 util.py 34
B.0.2 plotters.py 72
B.0.3 transFunc.py 74
B.1 Examples 81
B.1.1 CMWcheck.py 81
B.1.2 dg.py 85
C Solvers 87

C.1 2D_per.c 87
C.2 3D_per.c 99
Bibliography 115

List of Tables

Table Page

3.1 The Predicted Lamellae column represents simulations that took place with μ, σ and λ which should have generated lamellae as predicted by Choksi et al. whereas the Predicted Circles column represents simulations that took place with μ, σ and λ which should have generated hexagonally packed circles as predicted by Choksi et al. 28

List of Figures

Figure Page

3.1 A Double Gyroid 26

3.2 μ, λ and σ as in (3.1), $t = 15$. The simulation was carried out with a step-size of $h = 10^{-5}$ for $t \in [0, 14]$ and a step-size of $h = 10^{-6}$ for $t \in (14, 15]$. N was set to 64 for the simulation. Due to memory constraints, the nonlinearity was approximated using $2N$ Fourier coefficients instead of the $3N$ required to compute it exactly. 29

3.3 A typical energy decay for the 3D case with μ, λ and σ as in (3.1). 30

3.4 Hexagonally packed circles in the 2D case. 31

3.5 Lamellae in the 2D case. 32

Abstract

LONG TERM DYNAMICS OF THE DI-BLOCK COPOLYMER MODEL ON HIGHER DIMENSIONAL DOMAINS

Michael R. Atkins, M.S.

George Mason University, 2011

Thesis Director: Dr. Thomas Wanner

In this thesis, we examine the di-block copolymer model as proposed by Ohta and Kawasaki. We derive a nonlinear evolution equation from the Ohta-Kawasaki functional. We then find an approximation to this equation via Galerkin's method. A semi-implicit scheme is then applied to the Galerkin system. This solver is then implemented in C with a python user interface. This implementation is then used to investigate the long term dynamics of the model in 2D and 3D.

Specifically, we arrive at a solution to the 3D case which partially reproduces the results of Teramoto and Nishiura which describe the existence of a Double Gyroid equilibrium state. In the 2D case, we find a long term solution for many different parameter configurations. In fact, our results in the 2D case call in to question the efficacy of a nonstandard numerical method introduced by Choksi et al.

Chapter 1: Introduction

In this work, we examine a differential equation which is often used to model diblock copolymers. Block copolymers are molecules which are made up of two or more blocks of chemically distinct monomers each of which are either a small molecule or an atom[1]. Since we have restricted ourselves to diblock copolymers, the copolymers we will investigate only have two blocks, or monomers, per molecule. This class of copolymers exhibits phase separation when each monomer is viewed as a phase [1]. This phase separation is commonly modeled via the minimization of a free energy functional introduced by Ohta & Kawasaki which we will introduce more formally later [11].

Using this model, the phase diagram for this phenomenon has been partially constructed [2, 14]. Moreover, in attempting to describe the phase diagram, some new structures which have not been observed in experiment were observed in numerical simulations, namely, the double gyroid [14]. However, in spite of the promise of these results, a nonstandard numerical technique was used, and as a result, these results have not been independently reproduced. It is our goal to reproduce these results using a more standard numerical scheme. In order to achieve this aim, we will introduce a model for this phase separation, construct a numerical scheme for solving this model, discuss the implementation of this scheme and provide examples of solutions generated by this scheme.

1.1 Model

Let $\Omega \subset \mathbb{R}^3$ be the unit cube, and $u \in H^1_{\text{per}}(\Omega, [0, \infty))$ describe the ratio of monomers at a given point. The functional proposed by Ohta & Kawasaki which we will use is given by

$$E_{\lambda,\sigma}[u] = \int_\Omega \left[\frac{1}{2}|\nabla u|^2 + \lambda W(u) + \frac{\lambda\sigma}{2}|\nabla v|^2 \right] \, dx. \tag{1.1}$$

where λ is the reciprocal of the Flory–Huggins parameter [11],

$$W(u) := \frac{(1-u^2)^2}{4}, \quad -1 < \mu < 1, \quad \sigma > 0 \quad \text{and} \quad \lambda > 0, \tag{1.2}$$

u satisfies the average mass constraint

$$\mu := \int_\Omega u \, dx \tag{1.3}$$

and $v \in \left\{ f \in H^1_{\text{per}}(\Omega) \mid \int_\Omega f \, dx = 0 \right\}$ such that

$$-\Delta v = u - \mu \tag{1.4}$$

describes the long range interactions with σ scaling the strength of these interactions [3, 11].

It is our aim to find an evolution equation for $u(x,t)$. In order to do so, we apply a similar approach to that given in [3] and [5]. First, define the Hilbert space

$$H := \left\{ f \in L^2(\Omega) \;\middle|\; \int_\Omega f \, dx = 0 \right\}$$

equipped with the inner product

$$(f,g)_H := \int_\Omega \nabla u \nabla v \, dx \tag{1.5}$$

where $f = -\Delta u$ and $g = -\Delta v$ on Ω.

Second, define $\text{grad}_H E_{\lambda,\sigma}[u] \in H$ using the weak formulation of the H^{-1} norm, i.e., for all $\delta > 0$ and

$$B(t) : [0, \delta) \to H$$

such that B is differentiable at $t = 0$, $B(0) = u$ and B satisfies

$$\frac{d}{dt} E_{\lambda,\sigma}[B(t)]\Big|_{t=0} = \left(\mathrm{grad}_H E_{\lambda\sigma}[u], \frac{\partial B}{\partial t}\Big|_{t=0} \right).$$

Third, assume that u evolves along the path of steepest descent, i.e.,

$$\frac{\partial u}{\partial t} = -\mathrm{grad}_H E_{\lambda,\sigma}[u]. \tag{1.6}$$

Let u be as in (1.1) and $-\Delta w \in H$. Set $-\Delta v = u - \mu - t\Delta w$. We now compute

$$\begin{aligned} \frac{d}{dt} E_{\lambda,\sigma}[u - t\Delta w]\Big|_{t=0} = \Bigg[& \frac{d}{dt} \int_\Omega \frac{1}{2} |\nabla(u - t\Delta w)|^2 \, dx \\ & + \frac{d}{dt} \int_\Omega \lambda W(u - t\Delta w) \, dx \\ & + \frac{d}{dt} \int_\Omega \frac{\lambda\sigma}{2} |\nabla v|^2 \, dx \Bigg]_{t=0}. \end{aligned} \tag{1.7}$$

First, we find

$$\begin{aligned} \frac{d}{dt} \int_\Omega \frac{1}{2} |\nabla(u - t\Delta w)|^2 \, dx &= \frac{d}{dt} \int_\Omega \frac{1}{2} \nabla(u - t\Delta w) \cdot \nabla(u - t\Delta w) \, dx \\ &= \frac{d}{dt} \int_\Omega \frac{1}{2} \sum_{k=1}^d \left(\frac{\partial}{\partial x_k}(u - t\Delta w) \right)^2 \, dx \\ &= \int_\Omega \sum_{k=1}^d \left(\frac{\partial}{\partial x_k}(u - t\Delta w) \right) \left(\frac{d}{dt} \frac{\partial}{\partial x_k}(u - t\Delta w) \right) \, dx \\ &= \int_\Omega \sum_{k=1}^d \left(\frac{\partial}{\partial x_k}(u - t\Delta w) \right) \left(-\frac{\partial}{\partial x_k} \Delta w \right) \, dx. \end{aligned}$$

Evaluating at $t = 0$ yields

$$\begin{aligned}
\frac{d}{dt}\int_\Omega \frac{1}{2}|\nabla(u - t\Delta w)|^2\, dx\Big|_{t=0} &= \int_\Omega \sum_{k=1}^{d}\left(\frac{\partial}{\partial x_k}u\right)\left(-\frac{\partial}{\partial x_k}\Delta w\right)\, dx \\
&= -\int_\Omega \nabla u \cdot \nabla(\Delta w)\, dx \\
&= -\left(\int_{\partial\Omega} \Delta w \nabla u \cdot \boldsymbol{\nu}\, ds - \int_\Omega \Delta w \Delta u\, dx\right) \qquad (1.8) \\
&= \int_\Omega \Delta w \Delta u\, dx \qquad (1.9)
\end{aligned}$$

where (1.8) comes from Green's first identity and the equality in (1.9) comes from the periodic boundary conditions.

Now, we find

$$\begin{aligned}
\frac{d}{dt}\int_\Omega \lambda W(u - t\Delta w)\, dx\Big|_{t=0} &= \lambda \int_\Omega -\Delta w\, W'(u - t\Delta w)\, dx\Big|_{t=0} \\
&= -\lambda \int_\Omega W'(u)\Delta w\, dx. \qquad (1.10)
\end{aligned}$$

Finally, we find

$$\begin{aligned}
\frac{d}{dt}\int_\Omega \frac{\lambda\sigma}{2}|\nabla v|^2\,dx &= \frac{d}{dt}\frac{\lambda\sigma}{2}\int_\Omega \nabla v\cdot\nabla v\,dx \\
&= \frac{\lambda\sigma}{2}\int_\Omega \frac{d}{dt}\sum_{k=1}^{d}\left(\frac{\partial}{\partial x_k}v\right)^2\,dx \\
&= \lambda\sigma\int_\Omega \sum_{k=1}^{d}\left(\frac{\partial}{\partial x_k}v\right)\left(\frac{\partial}{\partial x_k}\frac{d}{dt}v\right)\,dx \\
&= \lambda\sigma\int_\Omega \sum_{k=1}^{d}\left(\frac{\partial}{\partial x_k}v\right)\left(\frac{\partial}{\partial x_k}w\right)\,dx \\
&= \lambda\sigma\int_\Omega \nabla v\cdot\nabla w\,dx.
\end{aligned}$$

Evaluating at $t=0$ yields

$$\frac{d}{dt}\int_\Omega \frac{\lambda\sigma}{2}|\nabla v|^2\,dx\Big|_{t=0} = \int_\Omega \lambda\sigma\nabla v\cdot\nabla w\,dx. \tag{1.11}$$

Combining (1.9), (1.10) and (1.11) with (1.7) yields

$$\begin{aligned}
\frac{d}{dt}E_{\lambda,\sigma}[u-t\Delta w]\Big|_{t=0} &= \int_\Omega \big[(\lambda W'(u)-\Delta u)(-\Delta w)+\sigma\lambda\nabla u\cdot\nabla w\,dx\big] \\
&= \int_\Omega \big[\nabla(\lambda W'(u)-\Delta u+\lambda\sigma v)\cdot\nabla w\,dx\big] \qquad (1.12) \\
&= \big(-\Delta(\lambda W'(u)-\Delta u+\lambda\sigma v),-\Delta w\big)_H \qquad (1.13)
\end{aligned}$$

where (1.12) follows from applying integration by parts and the boundary conditions, and

(1.13) comes from (1.5). Hence,

$$\begin{aligned}\mathrm{grad}_H E_{\lambda,\sigma}[u] &= -\Delta(\lambda W'(u) - \Delta u + \lambda\sigma v) \\ &= -\Delta(\lambda W'(u)) - \Delta u + \lambda\sigma(u-\mu). \end{aligned} \tag{1.14}$$

Combining this fact with our assumption in (1.6) yields

$$\frac{\partial u}{\partial t} = -\Delta^2 u - \lambda\Delta(u-u^3) - \lambda\sigma(u-\mu). \tag{1.15}$$

1.2 Galerkin's Method

In this section, we introduce Galerkin's method. We will use this to approximate the solution of (1.15), by finding an exact solution in a finite dimensional subspace of $H^1_{\mathrm{per}}(\Omega)$. It will be necessary to approximate the solution, since in general, even for linear PDE, only statements about existence of solutions can be made. First however, we must define what it means to solve (1.15).

Definition 1. *We say a function $u \in H^1_{per}(\Omega)$ is a solution to (1.15) provided that*

$$\left(-\Delta^2 u - \lambda\Delta(u-u^3) - \lambda\sigma(u-\mu) - \frac{\partial u}{\partial t}, v\right)_{L^2(\Omega)} = 0$$

for all $v \in H^1_{per}$ with $u(0) = u_0$.

With this definition for solutions in mind, we now provide Galerkin's method. Let $w_k(x)$ be a set of smooth functions which is a complete orthogonal set in $H^1_{\mathrm{per}}(\Omega)$ and a complete orthonormal set in $L^2(\Omega)$. For any N, we can construct u_N such that

$$u_N(x,t) := \sum_{k=0}^{N} a_k(t) w_k(x) \tag{1.16}$$

for $a_k(t) \in H^1([0,\infty))$ satisfy

$$a_k(0) = (u_0, w_k)_{L^2(\Omega)} \text{ for all } k \leq N \tag{1.17}$$

and

$$\left(-\Delta^2 u_N + \lambda \Delta W'(u_N) - \lambda\sigma(u_N - \mu) - \frac{\partial u_N}{\partial t}, w_k\right)_{L^2(\Omega)} = 0 \text{ for all } k \leq N. \tag{1.18}$$

The existence and uniqueness of these $a_k(t)$ is guaranteed by the following theorem.

Theorem 1. *Provided that $\{w_k\}$ is complete and orthogonal in $H^1_{per}(\Omega)$ and is complete and orthonormal in $L^2(\Omega)$, for any $N \in \mathbb{N}_0$, there exists a unique u_N in the form of (1.16) which satisfies (1.17) and (1.18).*

Proof. Assume u_N as in (1.16). For simplicity, assume $\mu = 0$. Since the inner product is linear in the first argument, for all $k \leq N$

$$\begin{aligned}\left(-\Delta^2 u_N - \lambda\Delta(u_N - u_N^3) - \lambda\sigma(u_N) - \frac{\partial u_N}{\partial t}, w_k\right)_{L^2(\Omega)} = &- \left(\Delta^2 u_N, w_k\right)_{L^2(\Omega)} \\ &+ \left(\lambda\Delta W'(u_N), w_k\right)_{L^2(\Omega)} \\ &- \left(\lambda\sigma(u_N), w_k\right)_{L^2(\Omega)} \\ &- \left(\frac{\partial u_N}{\partial t}, w_k\right)_{L^2(\Omega)}\end{aligned}$$

Note that for some $b_k(t)$, $W'(u_N) = \sum_{k=0}^{\infty} b_k(t) w_k(x)$. Let $P_M(W'(u_N)) := \sum_{k=0}^{M} b_k(t) w_k(x)$. Assume that for some $M \in \mathbb{N}$, $W'(u_N) = P_M(W'(u_N))$. Also, set $c_\ell = (\Delta^2 w_\ell, w_k)_{L^2(\Omega)}$ and

$d_\ell = (\Delta w_\ell, w_k)$.

$$
\begin{aligned}
-\left(\Delta^2 u_N, w_k\right)_{L^2(\Omega)} &= -\sum_{\ell=0}^{N} a_\ell(t) c_\ell \\
\left(\lambda \Delta W'(u_N), w_k\right)_{L^2(\Omega)} &= \lambda \sum_{\ell=0}^{M} b_\ell(t) d_\ell \\
-\left(\lambda \sigma(u_N), w_k\right)_{L^2(\Omega)} &= -\lambda \sigma a_k(t) \\
-\left(\frac{\partial u_N}{\partial t}, w_k\right)_{L^2(\Omega)} &= -a_k'(t).
\end{aligned}
$$

If we construct u_N so that it satisfies (1.17), elementary ordinary differential equations theory tells us that there are unique $a_k(t)$ which satisfy

$$
0 = -\sum_{\ell=0}^{N} a_\ell(t) c_\ell + \lambda \sum_{\ell=0}^{M} b_\ell(t) d_\ell - \lambda \sigma a_k(t) - a_k'(t)
$$

given the initial condition u_0 and setting $a_k(0)$ as in (1.17).

♦

1.3 Fourier Analysis

In the previous section we saw that finite linear combinations of orthonormal basis functions will solve (1.15). In this section we will choose our orthonormal basis as well as prove some facts about the Fourier coefficients of a function over this basis.

1.3.1 A Basis of Eigenfunctions

Since we are working with periodic boundary conditions on the unit cube, the set of functions we will choose is

$$
w_k(x) = e^{2\pi i \sum_{j=1}^{d} k_j x_j} \tag{1.19}
$$

where $(x_1, x_2, \ldots, x_d) = x \in \mathbb{R}^d$ and $(k_1, k_2, \ldots, k_d) = k \in \mathbb{N}_0^d$. We have chosen these functions because they are the eigenfunctions of the Laplacian over the unit cube with periodic boundary conditions. This will allow us to turn differentiation into multiplication by an eigenvalue which will make implementation of a solver in software possible. We prove this fact in the following theorem.

Theorem 2. *Let $-\Delta$ denote the Laplacian over the unit cube in $\mathbb{R}^d$ with periodic boundary conditions. The functions w_k are eigenfunctions of $-\Delta$.*

Proof. In order to show that w_k are eigenfunctions of $-\Delta$, we must show for all $k \in \mathbb{N}_0^d$ that $-\Delta w_k = a_k w_k$ for $a_k \in \mathbb{R}$ and that $w_k \in H^1_{\text{per}}(\Omega)$. First, we show the former statement.

To proceed, we compute the partial derivative

$$\begin{aligned}
\frac{\partial^2}{\partial x_\ell^2} w_k &= \frac{\partial}{\partial x_\ell}\left[\frac{\partial}{\partial x_\ell} e^{2\pi i \sum_{j=1}^d k_j x_j}\right] \\
&= \frac{\partial}{\partial x_\ell}\left[2\pi i k_\ell e^{2\pi i \sum_{j=1}^d k_j x_j}\right] \\
&= -4\pi^2 k_\ell^2 e^{2\pi i \sum_{j=1}^d k_j x_j} \qquad (1.20)
\end{aligned}$$

Now the computation of the Laplacian becomes straightforward.

$$\begin{aligned}
-\Delta w_k &= -\sum_{\ell=1}^d \frac{\partial^2}{\partial x_\ell^2} w_k \\
&= -\sum_{\ell=1}^d -4\pi^2 k_\ell^2 e^{2\pi i \sum_{j=1}^d k_j x_j} \qquad (1.21) \\
&= 4\pi^2 \sum_{\ell=1}^d k_\ell^2 e^{2\pi i \sum_{j=1}^d k_j x_j} \qquad (1.22)
\end{aligned}$$

where the equality in (1.21) comes from (1.20).

It remains to be shown that for all $k \in \mathbb{N}_0^d$, $w_k \in H^1_{\text{per}}(\Omega)$. We saw in (1.20) that w_k

has a continuous derivative, so $w_k \in H^1(\Omega)$. Therefore, all that we need to show is that w_k satisfies the periodic boundary conditions. Let $c \in \mathbb{Z}^d$.

$$\begin{aligned} w_k(x+c) &= e^{2\pi i \sum_{j=1}^d k_j(x_j+c_j)} \\ &= e^{2\pi i \sum_{j=1}^d k_j x_j} e^{2\pi i \sum_{j=1}^d k_j c_j} \\ &= e^{2\pi i \sum_{j=1}^d k_j x_j} \end{aligned} \tag{1.23}$$

where the equality in (1.23) comes from Euler's identity. ♦

In addition to proving the theorem, the above proof also gives us that the eigenvalue corresponding to w_k is

$$\kappa_k = 4\pi^2 \sum_{\ell=1}^d k_\ell^2. \tag{1.24}$$

As well as being the eigenfunctions of $-\Delta$, $\{w_k\}$ is a complete orthonormal set in $L^2(\Omega)$ and a complete orthogonal set in $H^1_{\text{per}}(\Omega)$ which we show in the next theorem.

Theorem 3. *$\{w_k\}$ is a complete orthonormal set in $L^2(\Omega)$ and a complete orthogonal set in $H^1_{per}(\Omega)$.*

Proof. From Theorem 2, we know that for all $k \in \mathbb{N}_0^d$, $w_k \in H^1_{\text{per}}(\Omega) \subset L^2(\Omega)$. First we show that $\{w_k\}$ is orthonormal in $L^2(\Omega)$. From the definition of the $L_2(\Omega)$ inner product, we have

$$\begin{aligned} (w_k, w_\ell)_{L^2(\Omega)} &= \int_\Omega w_k \overline{w_\ell}\, dx \\ &= \int_\Omega e^{2\pi i \sum_{j=1}^d k_j x_j} e^{2\pi i \sum_{j=1}^d \ell_j x_j}\, dx \\ &= \int_\Omega e^{2\pi i \sum_{j=1}^d (k_j - \ell_j) x_j}\, dx. \end{aligned} \tag{1.25}$$

If $k = j$, we have $(w_j, w_k)_{L^2(\Omega)} = 1$; otherwise, $(w_j, w_k)_{L^2(\Omega)} = 0$.

We now show that $\{w_k\}$ is orthogonal in $H^1_{\text{per}}(\Omega)$. Let $j \neq k$. From the definition of the $H^1_{\text{per}}(\Omega)$ inner product,

$$
\begin{aligned}
(w_k, w_\ell)_{H^1_{\text{per}}(\Omega)} &= (w_k, w_\ell)_{L^2(\Omega)} + \sum_{m=1}^{d} \left(\frac{\partial}{\partial x_m} w_k, \frac{\partial}{\partial x_m} w_\ell \right)_{L^2(\Omega)} \\
&= \sum_{m=1}^{d} \left(2\pi i k_m e^{2\pi i \sum_{j=1}^{d} k_j x_j}, 2\pi i \ell_m e^{2\pi i \sum_{j=1}^{d} \ell_j x_j} \right)_{L^2(\Omega)} \qquad (1.26) \\
&= \sum_{m=1}^{d} -4\pi^2 k_m \ell_m \int_\Omega e^{2\pi i \sum_{j=1}^{d} (k_j - \ell_j) x_j} \, dx \qquad (1.27) \\
&= 0
\end{aligned}
$$

where the equality in (1.26) comes from (1.25) and the equality in (1.27) comes from the definition of the inner product in $L^2(\Omega)$.

It remains to show that $\{w_k\}$ is complete in $H^1_{\text{per}}(\Omega)$ and $L^2(\Omega)$. This part of the proof is outside of the scope of this thesis, so we will simply cite [4] to complete the proof. ♦

We have now established that w_k satisfies the conditions for Theorem 1; hence, for any $N \in \mathbb{N}_0$, there is a unique u_N which approximates (1.15) such that

$$
u_N(x,t) = \sum_{|k|_\infty \leq N} a_k(t) w_k(x). \qquad (1.28)
$$

1.3.2 A Fact About Fourier Coefficients

One fact that we take advantage of in Section 2.2 pertains to the nonlinear term in (1.15), $u - u^3$.

Theorem 4.

$$u_N^3(x,t) = \sum_{|k|_\infty \le 3N} b_k(t) w_k(x)$$

for some $b_k : [0,\infty) \to \mathbb{R}$.

Proof. From (1.28), we have

$$\begin{aligned} u_N^3(x,t) &= \left(\sum_{|k|_\infty \le N} a_k(t) w_k(x) \right)^3 \\ &= \left(\sum_{|k|_\infty \le N} a_k(t) e^{2\pi i \sum_{j=1}^d k_j x_j} \right)^3 \qquad (1.29) \\ &= \sum_{|k|_\infty \le N} \sum_{|\ell|_\infty \le N} \sum_{|m|_\infty \le N} a_k(t) a_\ell(t) a_m(t) e^{2\pi i \sum_{j=1}^d k_j x_j} e^{2\pi i \sum_{\ell=1}^d k_j x_j} e^{2\pi i \sum_{m=1}^d k_j x_j} \\ &= \sum_{|k|_\infty \le N} \sum_{|\ell|_\infty \le N} \sum_{|m|_\infty \le N} a_k(t) a_\ell(t) a_m(t) e^{2\pi i \sum_{j=1}^d (k_j + \ell_j + m_j) x_j}. \qquad (1.30) \end{aligned}$$

Setting

$$b_n := \sum_{k+\ell+m=n} a_k(t) + a_\ell(t) + a_m(t) \qquad (1.31)$$

yields

$$\sum_{|k|_\infty \le N} \sum_{|\ell|_\infty \le N} \sum_{|m|_\infty \le N} a_k(t) a_\ell(t) a_m(t) e^{2\pi i \sum_{j=1}^d (k_j + \ell_j + m_j) x_j} = \sum_{|n|_\infty \le 3N} b_n(t) e^{2\pi i \sum_{j=1}^d n_j x_j}. \qquad (1.32)$$

♦

1.4 Semi Implicit Method

An important feature of (1.15) which was omitted in the preceding two sections is that of time. In this section, we will introduce a time stepping method for approximating a solution of an autonomous ODE and show that this method is convergent and is order 1. Of course, we must first define what we mean by convergent and order 1.

Let $u : [0, \infty) \to \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}^d$ such that for all $t \geq 0$,

$$u'(t) = f(u(t)) \tag{1.33}$$

where f is Lipschitz. We approximate u over $[0, t^*]$ by first choosing a discretization width $h > 0$ and using it to define $t_{n,h}$ via $t_{n+1,h} = t_{n,h} + h$, where $t_{0,h} = 0$. A time stepping method, F seeks to approximate $u(t_{n,h}) \approx u^{n,h}$ via $u^{n,h} = F(u^{n,h}, u^{n-1,h}, f)$[1].

Definition 2. *A time stepping method for the ODE (1.33) is convergent provided that for all $t^* > 0$,*

$$\lim_{h \to 0^+} \max_{n \in [0, \lfloor t^*/h \rfloor] \cap \mathbb{Z}} ||u^{n,h} - u(t_{n,h})|| = 0. \tag{1.34}$$

Definition 3. *A time stepping method F for the ODE (1.33) is order $p \geq 1$ provided for $h > 0, n \in \mathbb{N}$,*

$$u(t_{n,h}) - F(u(t_{n,h}), u(t_{n-1,h}), f)) = \mathcal{O}(h^{p+1}). \tag{1.35}$$

The time stepping method we will use requires a further constraint on f. Namely, that $f = f_1 + f_2$, for some f_1 and f_2 where f_1 is linear and f_2 is Lipschitz. This method is defined by

$$F(u^{n,h}, u^{n-1,h}, f) := u^{n-1,h} + hf_1(u^{n,h}) + hf_2(u^{n-1,h}). \tag{1.36}$$

[1]This definition is overly specialized for simplicity here; for a more general context, see [7].

In order for this method to be useful, it will need to be convergent and at least order $p = 1$ [7].

Theorem 5. *The time stepping method given in* (1.36) *is order 1.*

Proof. Let $h > 0$, $n \in \mathbb{N}$. Substituting (1.36) into the left side of (1.35) yields

$$\begin{aligned}
u(t_{n,h}) - F(u(t_{n,h}), u(t_{n-1,h}), f)) &= u(t_{n,h}) - u(t_{n-1,h}) \\
&\quad - hf_1(u(t_{n,h})) - hf_2(u(t_{n-1,h})) \\
&= u(t_{n-1,h}) + hu'(t_{n-1,h}) + \mathcal{O}(h^2) - u(t_{n-1,h}) \qquad (1.37) \\
&\quad - hf_1\left(u(t_{n-1,h}) + \mathcal{O}(h)\right) \\
&\quad - hf_2(u(t_{n-1,h})) \\
&= hu'(t_{n-1,h}) + \mathcal{O}(h^2) \qquad (1.38) \\
&\quad - hf_1(u(t_{n-1,h})) - hf_1(\mathcal{O}(h)) \\
&\quad - hf_2(u(t_{n-1,h})) \\
&= h(u'(t_{n-1,h}) - f(u(t_{n-1,h}))) + \mathcal{O}(h^2) \qquad (1.39) \\
&= \mathcal{O}(h^2), \qquad (1.40)
\end{aligned}$$

where the equality in step (1.37) comes from the Taylor series expansion of $u(t_{n,h})$ centered at $t_{n-1,h}$, the equality in step (1.38) comes from the linearity of f_1 and the equality in (1.40) comes from (1.33). ♦

Theorem 6. *The time stepping method given in* (1.36) *is convergent.*

Proof. We assume without loss of generality that f_2 is globally Lipschitz. Due to our choice of W in (1.2), given sufficient initial conditions, $|u|_\infty \leq M$ for some $M \in \mathbb{R}$, so f_2 can be

extended linearly on $(-\infty, -M) \cup (M, \infty)$ to satisfy globally Lipschitz.

Let $t^*, h > 0$. Define the $e^{n,h} := y^{n,h} - y(t_{n,h})$. From (1.36) we have

$$u^{n,h} = u^{n-1,h} + hf_1(u^{n,h}) + hf_2(u^{n-1,h}).$$

combining this with Theorem 5 yields

$$e^{n,h} = e^{n-1,h} + h(f_1(u^{n,h}) - f_1(u(t_{n,h}))) + h(f_2(u^{n-1,h}) - f_2(u(t_{n-1,h}))) + \mathcal{O}(h^2). \quad (1.41)$$

By the triangle inequality and the fact that $\mathcal{O}(h^2) \leq ch^2$ for some $c \in \mathbb{R}$,

$$||e^{n,h}|| \leq ||e^{n-1,h}|| + h||(f_1(u^{n,h}) - f_1(u(t_{n,h})))|| + h||(f_2(u^{n-1,h}) - f_2(u(t_{n-1,h})))|| + ch^2 \quad (1.42)$$

Since f_1 is linear and f_2 is globally Lipschitz, there exist $\lambda_1, \lambda_2 > 0$ such that for all $u_1, u_2 \in \mathbb{R}^d$,

$$||f_1(u_1) - f_1(u_2)|| \leq \lambda_1 ||u_1 - u_2|| \quad \text{and} \quad ||f_2(u_1) - f_2(u_2)|| \leq \lambda_2 ||u_1 - u_2||.$$

If λ_2 is a Lipschitz coefficient for f_2, $\lambda_2 + \varepsilon$ where $\varepsilon > 0$ is a Lipschitz coefficient for f_2 as well. Therefore, we can assume without loss of generality that $\lambda_1 \neq \lambda_2$. Applying this fact to (1.42) yields

$$||e^{n,h}|| \leq ||e^{n-1,h}|| + h\lambda_1 ||e^{n,h}|| + h\lambda_2 ||e^{n-1,h}|| + ch^2.$$

Since we will be taking a limit as $h \to 0^+$, we can assume without loss of generality $1 > h\lambda_1$. This gives us

$$||e^{n,h}|| \leq \frac{1 + h\lambda_2}{1 - h\lambda_1} ||e^{n-1,h}|| + \frac{ch^2}{1 - h\lambda_1}. \quad (1.43)$$

To proceed any further, we need to show that

$$||e^{n,h}|| \leq \frac{ch^2}{1-h\lambda_1} \frac{1}{\frac{1+h\lambda_2}{1-h\lambda_1} - 1} \left[\left(\frac{1+h\lambda_2}{1-h\lambda_1} \right)^n - 1 \right] \text{ for } n = 0, 1, \ldots. \tag{1.44}$$

We show this by induction. For $n = 0$, $u^{0,h} = u(0)$, so $||e^{0,h}|| = 0 \leq 0$; hence, the base case is true. Assume (1.44) for n. By (1.43),

$$\begin{aligned} ||e^{n+1,h}|| &\leq \frac{1+h\lambda_2}{1-h\lambda_1} \frac{ch^2}{1-h\lambda_1} \frac{1}{\frac{1+h\lambda_2}{1-h\lambda_1} - 1} \left[\left(\frac{1+h\lambda_2}{1-h\lambda_1} \right)^n - 1 \right] + \frac{ch^2}{1-h\lambda_1} \\ &= \frac{ch^2}{1-h\lambda_1} \frac{1}{\frac{1+h\lambda_2}{1-h\lambda_1} - 1} \left[\left(\frac{1+h\lambda_2}{1-h\lambda_1} \right)^{n+1} - \frac{1+h\lambda_2}{1-h\lambda_1} + \frac{1+h\lambda_2}{1-h\lambda_1} - 1 \right] \\ &= \frac{ch^2}{1-h\lambda_1} \frac{1}{\frac{1+h\lambda_2}{1-h\lambda_1} - 1} \left[\left(\frac{1+h\lambda_2}{1-h\lambda_1} \right)^{n+1} - 1 \right]. \end{aligned} \tag{1.45}$$

Hence, (1.44) is indeed the case for all n.

Moreover, we have that

$$\begin{aligned} \left(\frac{1+h\lambda_2}{1-h\lambda_1} \right)^n &= \left(\frac{1-h\lambda_1+h\lambda_1+h\lambda_2}{1-h\lambda_1} \right)^n \\ &= \left(1 + \frac{h\lambda_1+h\lambda_2}{1-h\lambda_1} \right)^n \\ &\leq \exp \left(\frac{h\lambda_1+h\lambda_2}{1-h\lambda_1} \right)^n \\ &= \exp \left(\frac{nh\lambda_1+nh\lambda_2}{1-h\lambda_1} \right). \end{aligned} \tag{1.46}$$

Since $n \in [0, \lfloor t^*/h \rfloor] \cap \mathbb{Z}$, for all such n,

$$\exp \left(\frac{nh\lambda_1+nh\lambda_2}{1-h\lambda_1} \right) \leq \exp \left(\frac{t^*\lambda_1+t^*\lambda_2}{1-h\lambda_1} \right). \tag{1.47}$$

To complete the proof, we must show that

$$\lim_{h\to 0} \frac{ch^2}{1-h\lambda_1} \frac{1}{\frac{1+h\lambda_2}{1-h\lambda_1}-1} \left[\exp\left(\frac{t^*\lambda_1 + t^*\lambda_2}{1-h\lambda_1}\right) - 1\right] = 0. \tag{1.48}$$

First, we note that

$$\lim_{h\to 0} \exp\left(\frac{t^*\lambda_1 + t^*\lambda_2}{1-h\lambda_1}\right) - 1 = \exp\left(t^*\lambda_1 + t^*\lambda_2\right) - 1. \tag{1.49}$$

Now we must handle the other term in (1.48).

$$\begin{aligned}
\lim_{h\to 0} \frac{ch^2}{(1-h\lambda_1)\left(\frac{1+h\lambda_2}{1-h\lambda_1}-1\right)} &= \lim_{h\to 0} \frac{ch^2}{(1+h\lambda_2)-(1-h\lambda_1)} \\
&= \lim_{h\to 0} \frac{ch^2}{h(\lambda_2-\lambda_1)} \\
&= \lim_{h\to 0} \frac{ch}{\lambda_2-\lambda_1} \\
&= 0 \qquad (1.50)
\end{aligned}$$

where (1.50) comes from choosing λ_1 and λ_2 so that $\lambda_2 \neq \lambda_1$. ♦

Chapter 2: Numerical Scheme

In this chapter, we will use the results built in the previous chapter to derive a numerical scheme for solving (1.15) numerically. Once we have developed a scheme, we will discuss how the scheme was implemented in software as well as a software package developed to ease the use of the solvers which implement the scheme.

2.1 Derivation

Let κ_ℓ and w_ℓ denote the eigenvalues and eigenfunctions of the Laplacian operator over the unit cube in $\mathbb{R}^3$ with periodic boundary conditions. In Section 1.3.1 we saw that these functions satisfied the hypothesis of Theorem 1 which yielded an approximate solution to (1.15), (1.28). This approximation is still exact in t, so here we apply the time stepping method (1.36) to approximate in t as well.

To this end, we must first discretize t. Pick h to be the discretization width. We discretize the interval $[0, T]$ by $D_{T,h} := [0, \lceil T/h \rceil] \cap \mathbb{Z}$. Given $N \in \mathbb{N}_0$, $k \in D_{T,h}$, let $u_N^k \approx u_N(kh)$ via some time stepping method.

In order to use the time stepping method (1.36), we must first split the right side of (1.15) into a linear part f_1 and a Lipschitz part f_2. Define

$$f_1(u_N^{k+1}) = -\Delta(\Delta u_N^{k+1}) - \lambda\sigma(u_N^{k+1} - \mu)$$

$$f_2(u_N^k) = \lambda\Delta((u_N^k)^3 - u_N^k)$$

From the definition of u_N^k, (1.28) and Theorem 4, for all $k \in \mathbb{N}_0$, there exist a_ℓ^k and b_ℓ^k such

that

$$u_N^k = \sum_{|\ell|_\infty \le N} a_\ell^k w_\ell, \text{ and } (u_N^k)^3 - u_N^k \approx \sum_{|\ell|_\infty \le N} b_\ell^k w_\ell. \tag{2.1}$$

The fact that the w_ℓ are eigenfunctions couples with (2.1) to yield

$$\begin{aligned} f_1(u_N^{k+1}) &= -\Delta\left(\Delta \sum_{|\ell|_\infty \le N} a_\ell^{k+1} w_\ell\right) - \lambda\sigma\left(\sum_{|\ell|_\infty \le N} a_\ell^{k+1} w_\ell - \mu\right) \\ &= -\sum_{|\ell|_\infty \le N} \kappa_\ell^2 a_\ell^{k+1} w_\ell - \lambda\sigma\left(\sum_{|\ell|_\infty \le N} a_\ell^{k+1} w_\ell - \mu\right) \end{aligned} \tag{2.2}$$

$$\begin{aligned} f_2(u_N^k) &= \lambda\Delta((u_N^k)^3 - u_N^k) \\ &= \lambda\Delta\left(\sum_{|\ell|_\infty \le N} b_\ell^k w_\ell\right) \\ &= \lambda \sum_{|\ell|_\infty \le N} \kappa_k b_\ell^k w_\ell. \end{aligned} \tag{2.3}$$

We now apply the semi-implicit method (1.36) to u_N^k.

$$\begin{aligned} u_N^{k+1} &= u_N^k + hf_1(u_N^{k+1}) + hf_2(u_N^k) \\ \sum_{|\ell|_\infty \le N} a_\ell^{k+1} w_\ell &= \sum_{|\ell|_\infty \le N} a_\ell^k w_\ell - h\left[\sum_{|\ell|_\infty \le N} \kappa_\ell^2 a_\ell^{k+1} w_\ell + \lambda\sigma\left(\sum_{|\ell|_\infty \le N} a_\ell^{k+1} w_\ell - \mu\right)\right] \\ &\quad + h\left[\lambda \sum_{|\ell|_\infty \le N} \kappa_\ell b_\ell^k w_\ell\right] \end{aligned} \tag{2.4}$$

where (2.4) comes from substituting (2.1), (2.2) and (2.3). Since the w_ℓ are orthonormal,

we can look at each term within the summation individually. For $\ell \neq \mathbf{0}$, we have

$$a_\ell^{k+1} = a_\ell^k - h\left[\kappa_\ell^2 a_\ell^{k+1} + \lambda\sigma a_\ell^{k+1}\right] + h\lambda\kappa_\ell b_\ell^k$$

$$a_\ell^{k+1} = \frac{a_\ell^k + h\lambda\kappa_\ell b_\ell^k}{1 + h\kappa_\ell^2 + h\lambda\sigma}. \tag{2.5}$$

In the case that $\ell = \mathbf{0}$, we must refer back to the integral constraint (1.3). For all $k \in \mathbb{N}_0$, we have

$$\begin{aligned}
\mu &= \int_\Omega u_N^k \\
&= \int_\Omega \sum_{|\ell|_\infty \leq N} a_\ell^k w_\ell \qquad (2.6)\\
&= \sum_{|\ell|_\infty \leq N} a_\ell^k \int_\Omega w_\ell \\
&= a_{\mathbf{0}}^k \qquad (2.7)
\end{aligned}$$

where the equality in (2.6) comes from (2.1) and the equality in (2.7) comes from the fact that

$$\int_\Omega w_\ell = \begin{cases} 1 & \text{for } \ell = \mathbf{0} \\ 0 & \text{otherwise.} \end{cases}$$

2.2 Implementation in C

In this section we discuss how the numerical scheme in section 2.1 was implemented in software. We chose the C programming language to accomplish this as C allows one to write code that is both efficient and readable. Below is the algorithm which was implemented.

1. Compute a random initial condition via the Mersenne Twister [10].

2. Compute the fast Fourier transform of this initial condition and store it as a_ℓ^0.

3. Compute b_ℓ^k from a_ℓ^k for each ℓ via the fast Fourier transform.

4. Compute a_ℓ^{k+1} for each ℓ via (2.5).

5. Goto Step 3 until k is sufficiently large.

In order to use the fast Fourier transform, we must first discretize our domain. Given N, we discretize the d-dimensional unit cube by $X = x_\ell$ where, for all $\ell \in \{0, 1, \ldots, N-1\}^d$,

$$x_\ell = \left(\frac{\ell_0}{N}, \frac{\ell_1}{N}, \ldots, \frac{\ell_d}{N}\right). \tag{2.8}$$

This will allow us to describe what we mean when we say compute the fast Fourier transform in the next section. To see how this algorithm was implemented, see Appendix C.

2.2.1 Fast Fourier Transform

Here we describe how we use the fast Fourier transform to compute the b_ℓ^k in part 3 of the above algorithm. The data we store are the Fourier coefficients of u_N^k, i.e., a_ℓ^k. In order to compute $(u_N^k)^3 - u_N^k$, we must first transform the a_ℓ^k to $u_N^k(x_\ell)$. Which we do for $\ell \in \{0, 1, \ldots, 3N-1\}$ as it will allow us to compute $(u_N^k)^3$ exactly due to Theorem 4. This is done using the Fastest Fourier Transform in the West library (FFTW) which computes the inverse Fourier transform, i.e., for $m \in \{0, 1, \ldots, 3N-1\}^d$ [6]

$$\begin{aligned} Y_m &= \sum_{|\ell|_\infty < 3N} a_\ell^k e^{2\pi i/3N \sum_{j=1}^d m_j \ell_j} \\ &= \sum_{|\ell|_\infty < 3N} a_\ell^k e^{2\pi i \sum_{j=1}^d x_{m_j} \ell_j} \\ &= u_N^k(x_m). \end{aligned} \tag{2.9}$$

We then use FFTW to compute the Fourier transform of $(u_N^k(x_m))^3 - u_N^k(x_m)$ to obtain b_ℓ^k [6].

$$
\begin{aligned}
X_\ell &= \sum_{|m|_\infty < 3N} ((u_N^k(x_m))^3 - u_N^k(x_m)) e^{-2\pi i/3N \sum_{j=1}^d m_j \ell_j} \\
&= \sum_{|m|_\infty < 3N} b_\ell^k w_\ell(x_m) e^{-2\pi i/3N \sum_{j=1}^d m_j \ell_j} \qquad (2.10) \\
&= \sum_{|m|_\infty < 3N} b_\ell^k e^{2\pi i \sum_{j=1}^d (x_{m_j} \ell_j) - 2\pi i/3N \sum_{j=1}^d m_j \ell_j} \\
&= \sum_{|m|_\infty < 3N} b_\ell^k \\
&= (3N)^d b_\ell^k \qquad (2.11)
\end{aligned}
$$

where the equality in (2.10) comes from Theorem 4.

In this way we are able to leverage FFTW to compute our nonlinearity. However, we have glossed over how we managed to sum a_ℓ^k over $\ell \in \{0, 1, \ldots, 3N-1\}^d$ when we have only defined a_ℓ^k for $\ell \in \{0, 1, \ldots, N-1\}^d$. Unfortunately, due to a symmetry condition given for the fast Fourier transform of real data, we can't simply append zeros to the places where a_ℓ^k is not defined [6]. That would break the symmetry condition, so we need to pad a_ℓ^k more carefully which is discussed in section 2.2.2.

2.2.2 Zero Padding

In this section we describe how the C array which contained the a_ℓ^k was padded with zeros so to preserve symmetry. However, we must first describe how data is input and output in FFTW for transforms of real data.

For the inverse Fourier transform, FFTW is given an array of $N^{d-1}(N/2+1)$ complex numbers and an array of N^d double precision numbers. The inverse Fourier transform of the array of complex numbers is then computed and stored in the array of double precision

numbers. For the Fourier transform, the same arrays are given, but the Fourier transform of the array of double precision numbers is computed and stored in the array of complex numbers. The reason FFTW only needs an array of $N^{d-1}(N/2+1)$ complex numbers instead of N^d is because FFTW uses the symmetry condition to determine the missing entries [6].

The symmetry condition can be thought of as each element of the array has an element in the opposite corner which is its conjugate, and is a result of the fact that u must be real [6]. More precisely, define

$$f(n) := \begin{cases} 0 & \text{if } n = 0 \\ N - n & \text{otherwise} \end{cases}$$

The symmetry condition is given by

$$a_{(\ell_1,\ell_2,\ldots,\ell_d)} = \overline{a_{(f(\ell_1),f(\ell_2),\ldots,f(\ell_d))}}. \tag{2.12}$$

If (2.12) is broken, then the result of the transform will no longer be real, so in order to guarantee that this condition is not broken, we map corners to corners and place zeros everywhere else. To see this stated more precisely, please see Appendix C, specifically, the function `a_copy`.

2.3 Implementation in Python

Here we describe a Python software package which was developed to provide an easy to use interface to the C solver discussed in section 2.2. The goal was to have an interface which would provide the following features.

1. Automated plotting of results

2. Save Run to disk

3. Restart runs

4. Start multiple runs at once and take advantage of all available processors

Python was chosen since features 2 and 4 could easily be accomplished using the Python standard library, and feature 1 could be achieved via the matplotlib and pyvtk libraries [8, 12, 13]. Also, by choosing Python, we are able to easily document and test the code by using doctests to see this, please look to appendix B [13].

In order to achieve features 2 and 3, a `Path` class was created with attributes which would contain all the data generated by a solver such as the Fourier coefficients, energies and norms. This class also contains a save method which uses pickling, a part of the Python standard library, to save the entire object. In order to restart a run, this object is simply reloaded, has its `timeEnd` attribute increased and the run continued via the `evolve` method. Feature 1 was accomplished by looping through the data saved in the `Path` object and using either matplotlib or pyvtk to save images or vtk files of the plots generated by the data, respectively. To achieve feature 4, the `runBuilder` method was created. This method essentially takes a list of python dictionaries which contain parameters for runs which should be performed. Then, the program uses the multiprocessing library to divide these runs over several processes. Each of these processes then creates a subprocess which runs the solver with the desired parameters, saves the created `Path` object and plots the results. In this way, one is able to give the computer a large amount of instructions for performing runs with very short scripts. To see examples of such scripts, please see appendix B.1.

Chapter 3: Results and Conclusions

Now that we have implemented the scheme derived in section 2.1, we can use it to examine some previously obtained results from the literature. Specifically, we will attempt to create a double gyroid in the 3D case as in Teramoto and Nishiura, and examine the phase diagram for the 2D case presented by Choksi et al. [2, 14].

3.1 Creating a Double Gyroid

In their 2002 paper, Teramoto and Nishiura present results which confirm the existence of a double gyroid (see Figure 3.1) structure as an equilibrium phase configuration under certain choices of μ, λ, σ [14]. Unfortunately, the numerical scheme used to generate these results was a nonstandard technique they were not independently reproduced using standard numerical techniques. We have attempted to reproduce the double gyroid structure with the following parameters taken from [15].

$$\mu = 0.2, \lambda = \frac{2}{(0.03)^2} \text{ and } \sigma = 0.03 \cdot 2^{11}(1-\mu^2)^{-2}. \tag{3.1}$$

Unfortunately, due to stability constraints and the inherently long runtime (approximately 24 hours per simulation time unit) necessary to obtain solutions to 3D questions, we are only able to produce the intermediate structure, Figure 3.2 which looks close enough to a double gyroid that we can hope that this simulation will result in a double gyroid equilibrium state. The runtime issue was compounded by the fact that in this model, suboptimal states have energy which is very close to the energy of optimal states [3], so a long simulation time is necessary to reach the stable equilibrium state. This fact can be seen in the energy decay, Figure 3.3, which after a very rapid initial decay, flattens out dramatically. However, it is

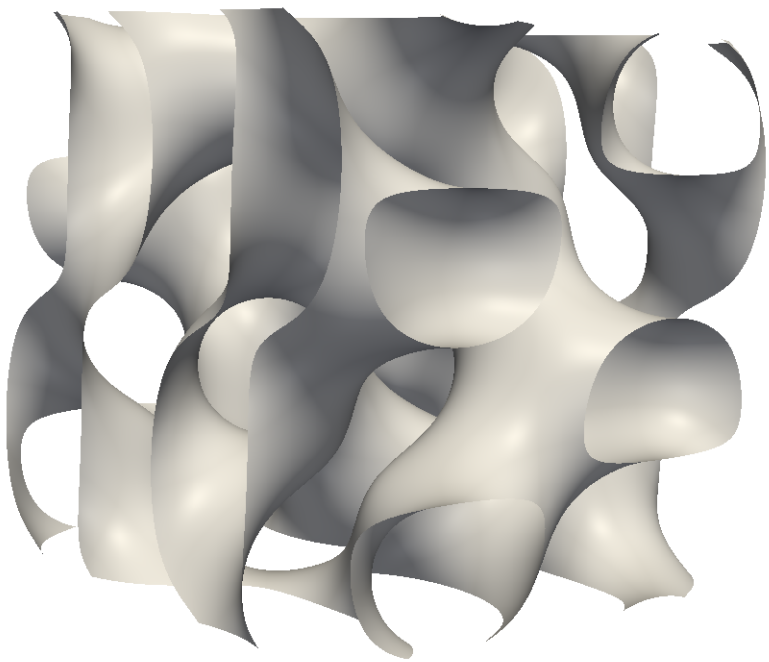

Figure 3.1: A Double Gyroid

also possible that the approximation created by the Galerkin system we use is leading our solution into an equilibrium for our approximate system that is not an equilibrium for the physical system. Future work would need to verify that this is not the case.

3.2 Examining a Phase Diagram

In the 2D case of the model, given μ, λ and $\sigma = 1$, there is either one of two stable equilibrium solutions or no stable solution. These two stable states are hexagonally packed circles and lamellae (see Figures 3.4 and 3.5). In a paper recently submitted by Choksi, et al., they

make the following statement about the phase diagram for the 2D problem [2].

Define $\gamma := \sqrt{\lambda}$, and set $\sigma = 1$. If

$$\gamma = \frac{2}{1 - 3\frac{\mu^2}{\beta^2}}, \tag{3.2}$$

the lamellae state is stable provided

$$0 \leq \beta < \frac{1}{29}\sqrt{551 - 174\sqrt{6}} \tag{3.3}$$

and the hexagonally packed circular state is stable provided

$$\frac{1}{29}\sqrt{551 - 174\sqrt{6}} < \beta < 3\sqrt{\frac{5}{37}}. \tag{3.4}$$

This result is accompanied by some numerical simulations which verify the result; however, these numerical simulations were done with a nonstandard scheme where noise was added to coerce solutions away from unstable equilibria [2].

In order to evaluate the efficacy of this scheme, we performed simulations using randomly chosen λ and μ which satisfied either

$$\beta = -.001 + \frac{1}{29}\sqrt{551 - 174\sqrt{6}},$$

$$\beta = .001 + \frac{1}{29}\sqrt{551 - 174\sqrt{6}},$$

$$\text{or } \beta = -.001 + 3\sqrt{\frac{5}{37}}, \tag{3.5}$$

as well as (3.2). σ was set to 1 and N was set to 40 with $3N$ Fourier coefficients used to compute the nonlinearity. In order to recreate the results of Choksi et al., we changed our

	Predicted Lamellae	Predicted Circles
Actual Lamellae	8	9
Actual Circles	4	5

Table 3.1: The Predicted Lamellae column represents simulations that took place with μ, σ and λ which should have generated lamellae as predicted by Choksi et al. whereas the Predicted Circles column represents simulations that took place with μ, σ and λ which should have generated hexagonally packed circles as predicted by Choksi et al.

domain from the unit square to the L-square with $L = 4\pi$ for $\sqrt{2} \leq \lambda < \sqrt{10}$, $L = 2\pi$ for $\sqrt{10} \leq \lambda < 5$ and $L = \pi$ for $\lambda > 5$. This was accomplished by rescaling λ, σ and t. Since the solver discussed in section 2.2 is relatively efficient, we were able to compute solutions at $t = \frac{100000}{\lambda}$ with a step size of $h = \frac{0.1}{\lambda(1+\lambda^{\frac{3}{4}})}$ which removed the need for using a nonstandard scheme to find stable solutions. The results of these simulations are summarized in Table 3.1. Based on these results, it seems as if the numerical scheme used in Choksi et al. had some effect on the results presented there in the sense that the scheme coerced solutions into unstable equilibria.

3.3 Conclusion

In this thesis, we introduced the Ohta-Kawasaki functional, used a gradient flow argument to derive an evolution equation for this functional, developed a numerical scheme for solving this evolution equation, implemented this scheme in software and used this implementation to examine a pair of previous results. The results of Teramoto and Nishiura were partially verified, but due to the shallow energy decay could not be completely verified, while the results of Choksi et al. were shown to have some dependence on the numerical scheme used.

In future work, it would be interesting to see the results of Teramoto and Nishiura fully verified as well as the results of Choksi et al. more fully explored.

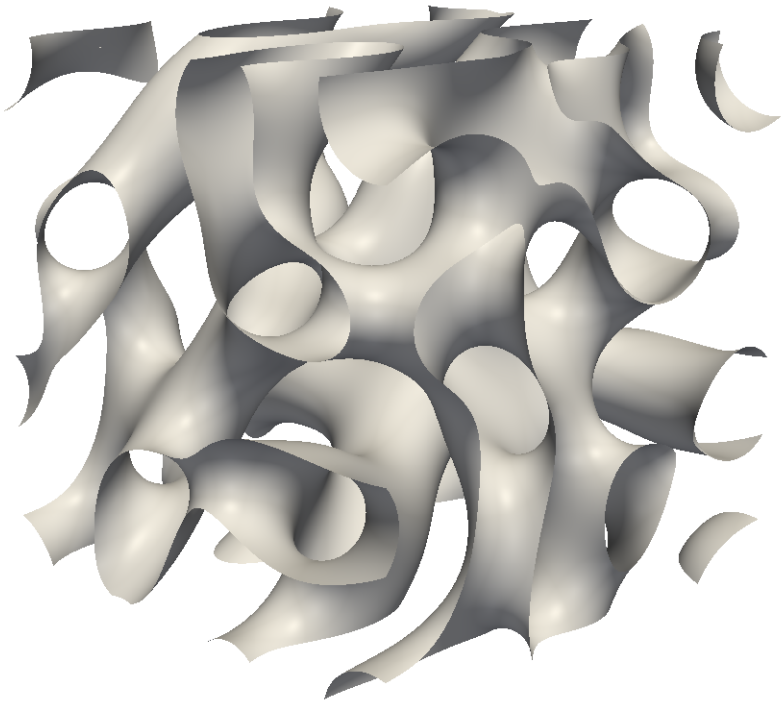

Figure 3.2: μ, λ and σ as in (3.1), $t = 15$. The simulation was carried out with a step-size of $h = 10^{-5}$ for $t \in [0, 14]$ and a step-size of $h = 10^{-6}$ for $t \in (14, 15]$. N was set to 64 for the simulation. Due to memory constraints, the nonlinearity was approximated using $2N$ Fourier coefficients instead of the $3N$ required to compute it exactly.

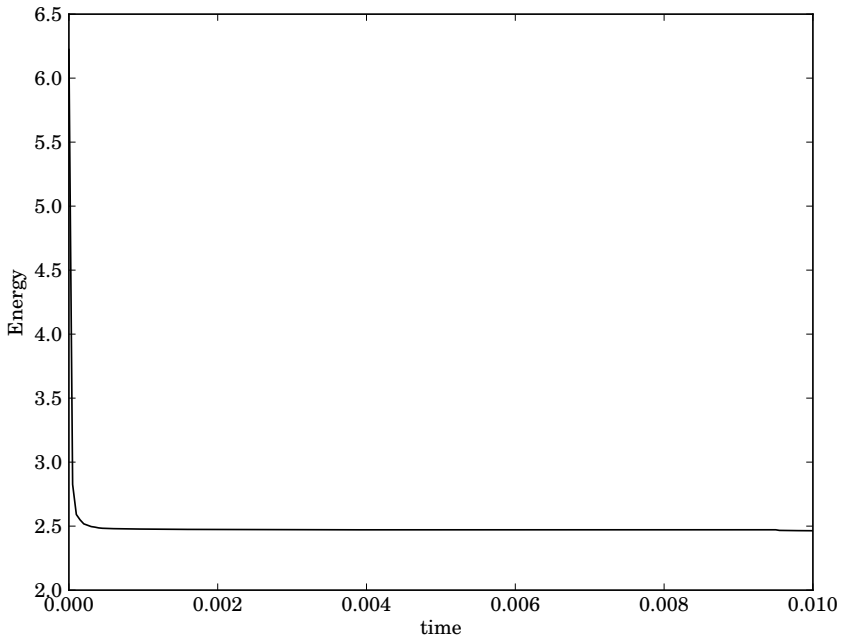

Figure 3.3: A typical energy decay for the 3D case with μ, λ and σ as in (3.1).

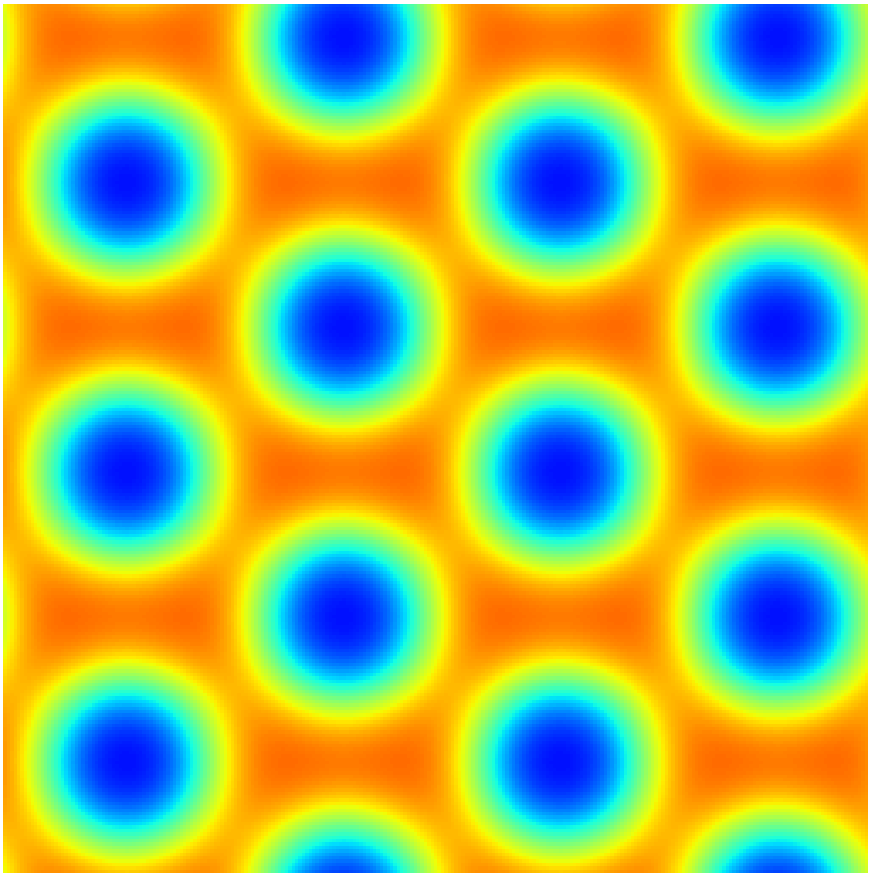

Figure 3.4: Hexagonally packed circles in the 2D case.

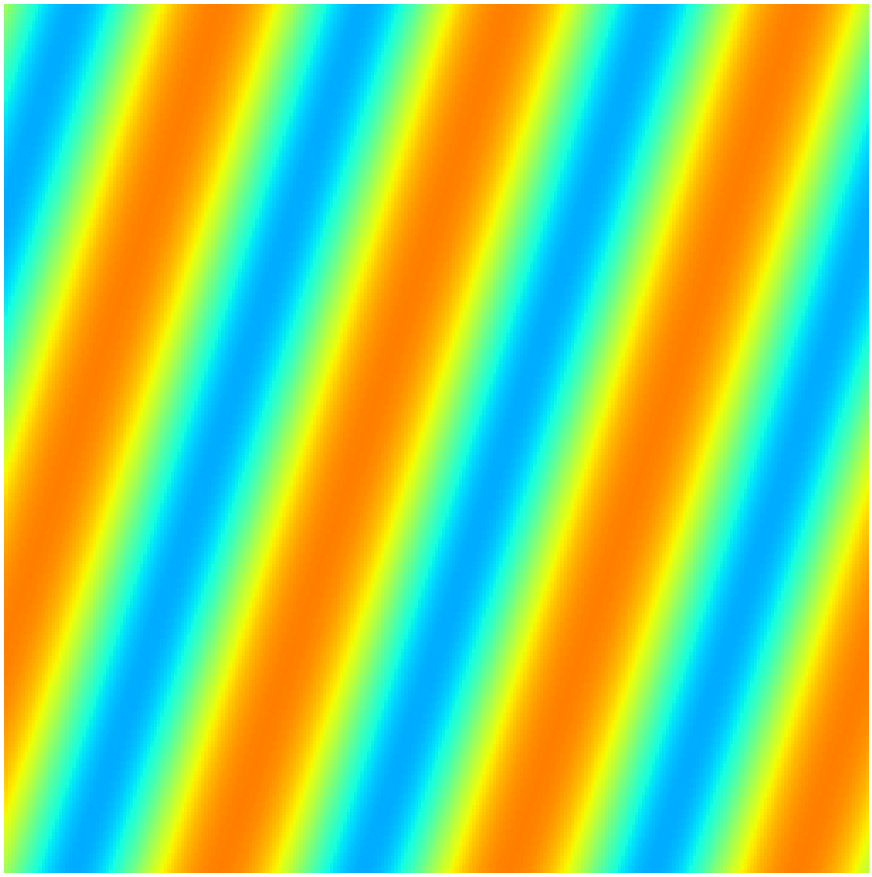

Figure 3.5: Lamellae in the 2D case.

Appendix A: Notation

- $\mathbb{N}_0$ The set $\{0, 1, 2, \dots\}$.
- $|\cdot|$ The Euclidean norm in $\mathbb{R}^d$
- $|\cdot|_\infty$ The max norm in $\mathbb{R}^d$.
- $||\cdot||_B$ The norm in the Banach space B.
- $L^2(\Omega)$ The space of functions which are square integrable in the Lebeguse sense.
- $H^1_{\text{per}}(\Omega)$ The space of functions which are weakly differentiable and periodic over Ω.
- Δ The Laplacian operator.
- $\overline{x}$ The complex conjugate of x.
- $(\cdot, \cdot)_H$ The inner product in the Hilbert space H.
- $\mathbf{0}$ The zero vector in some space (the space should be clear from the context).

Appendix B: AISE

B.0.1 util.py

```
import numpy, transFunc, plotters
import sys, subprocess, cPickle, gzip, os, copy, multiprocessing,
    functools
from math import sqrt, pi
from itertools import product

# These dictionaries map the BCs specified to the functions which
    are pertient
BC_funcs  = {'neu1' : transFunc.matlab_dct,
             'neu2' : transFunc.dct2,
             'neu3' : transFunc.dct3,
             'per1' : numpy.fft.rfft,
             'per2' : numpy.fft.rfft2,
             'per3' : numpy.fft.rfftn,
             'dir1' : transFunc.matlab_dst,
             'dir2' : transFunc.dst2
            }
BC_ifuncs = {'neu1' : transFunc.matlab_idct,
             'neu2' : transFunc.idct2,
             'neu3' : transFunc.idct3,
             'per1' : numpy.fft.irfft,
             'per2' : numpy.fft.irfft2,
```

```
                'per3' : numpy.fft.irfftn,
                'dir1' : transFunc.matlab_idst,
                'dir2' : transFunc.idst2
                }

class EqSoln(object):
    """A class which represents an equilibrium solution to the diblock
    copolymer model. When instantiated, the eigenfunctions will automatically
    be computed and stored.

    The initial perturbed solutions should be mass preserving
    >>> eqn = EqSoln(numpy.array([0,0]),lambd=400.,sigma=0.,mu=0.)
    >>> constant = 0.
    >>> for path in eqn.paths:
    ...     constant += numpy.abs(path.initial[0])
    >>> constant == 0.
    True
    """

    def __init__(self, soln, lambd, sigma, mu, BCs='neu1',
                 perturb=.001, solver='neu1', paths = False):
        self._soln      = soln
```

```
        self._lambda   = lambd
        self._sigma    = sigma
        self._mu       = mu
        self._perturb  = perturb
        self._solver   = solver
        self._BCs      = BCs

        if not paths:
            self._computeEigenfunctions()
        else: self._paths = copy.deepcopy(paths)

    @property
    def soln(self):
        """The equilibrium solution"""
        return self._soln.copy()

    @property
    def lambd(self):
        """The lambda value for this equilibrium solution"""
        return self._lambda

    @property
    def sigma(self):
        """The sigma value for this equilibrium solution"""
        return self._sigma
```

```python

    @property
    def mu(self):
        """The mu value for this equilibrium solution"""
        return self._mu

    @property
    def perturb(self):
        """The pertubation factor for computing the perturbed equilibrium
        solutions"""
        return self._perturb

    @property
    def solver(self):
        """The path to the timestepper"""
        return self._solver

    @solver.setter
    def solver(self, value):
        self._solver = value
        for evolution in self._eigenfunctions:
            evolution.solver = value

    @property
```

```
    def paths(self):
        """A list containing the unstable paths created by the
        eigenvalues/functions of the equilibrium solution."""

        return copy.copy(self._paths)

    @property
    def BCs(self):
        """The Boundary conditions. Available choices:
            'neu1'
            'neu2'
            'neu3'
            'per1'
            'per2'
            'per3'
            'dir1'
            'dir2'"""
        return self._BCs

    def _computeEigenfunctions(self):

        self._paths = []
        dot = numpy.dot
        diag = numpy.diag
        eig = numpy.linalg.eig
```

```
        soln = self.soln
        sigma = self.sigma
        lambd = self.lambd

        N = soln.shape[0]
        v = numpy.eye(N)
        A = numpy.zeros([N,N], numpy.complex128)

        ft = BC_funcs[self.BCs]
        rt = BC_ifuncs[self.BCs]

        for j in range(N):
            A[:,j] = 3*N*ft(((rt(soln))**2)\
                *(rt(v[:,j])))

        kappa = pi**2 * numpy.linspace(1,N,N)**2
        mat = -diag(kappa**2,0) + lambd * diag(kappa,0)\
              -lambd*dot(diag(kappa,0),A)\
              -sigma * v
        D,V = eig(mat)

        for i in range(len(D)):
            #Only examine unstable equilibria
            if D[i] <= 0: continue

```

```
            eigfunc = V[:,i]
            eigfunc[0] = 0.0
            eigval  = D[i]
            perturbedEigfunc = self._perturb * eigfunc + self._soln
            self._paths.append(Path(initial=perturbedEigfunc,
                                    eigval=eigval,
                                    BCs=self.BCs))

    def evolve(self, force=False):
        """This function calls the evolve method for the Paths which have not
        been previously evolved. The evolve method is called in parallel on as
        many processors that are available. If force is True, all Paths in
        self.paths are evolved."""

        paths = self._paths

        unsolved = []
        for path in paths:
            if force:
                path.solved = False
            if not path.solved:
```

```
                unsolved.append(path)

        for path in unsolved:
            paths.remove(path)

        ncpus  = multiprocessing.cpu_count()
        pool   = multiprocessing.Pool(processes=ncpus)
        evolver = lambda path : path.evolve()
        solved = pool.map(evolver,unsolved)
        paths += solved

    def plot(self, rootdir):
        """Recursively plot the paths of this equilibrium
           solution in the given
        root directory."""

        paths = self.paths

        for i in range(len(paths)):
            filepath = rootdir + '/%d' % i
            paths[i].plot(filepath)

    def save(self, filename):
        """Save this EqSoln to a file."""
        outfile = gzip.open(filename + '.gz', 'w')
```

```
        cPickle.dump(self, outfile)
        outfile.close()

class Path(object):
    """A class which will store the information about the evolution of some
    initial condition.

    An example of path creation with 2D periodic boundary conditions. Note that
    numPlots specifies the number of snapshots to generate other than the final
    configuration. In this case, the two snapshots generated are the initial
    condition and the final solution.
    >>> path = Path(solver   = 'per2',
    ...             lambd    = 1.0,
    ...             sigma    = 1.0,
    ...             mu       = .1,
    ...             stepSize = 1e-2,
    ...             BCs      = 'per2',
    ...             numPlots = 1,
    ...             timeEnd  = .1
    ...             )
    >>> path.evolve() #doctest: +ELLIPSIS
```

```
<__main__.Path object at 0x...>
>>> len(path.evolution)
2

Another example; this time with 3D periodic boundary conditions.
>>> path = Path(solver    = 'per3',
...                 lambd     = 1.0,
...                 sigma     = 1.0,
...                 mu        = .1,
...                 stepSize  = 1e-2,
...                 BCs       = 'per3',
...                 numPlots  = 1,
...                 timeEnd   = .1
...                 )
>>> path.evolve() #doctest: +ELLIPSIS
<__main__.Path object at 0x...>
>>> len(path.evolution)
2

The energies are computed the first time they are needed, and there will be
the same amount of energies as there will snapshots in evolution
>>> path.computeEnergies()
```

```
>>> len(path.energies) == len(path.evolution)
True

Also, the times are computed when the energies are computed, for ease when
plotting energies.
>>> len(path.energies) == len(path.times)
True

Also, if the number of plots exceeds the amount of time steps that will
occur, the number of time steps should be used instead.
>>> path = Path(solver   = 'per2',
...          lambd    = 1.0,
...          sigma    = 1.0,
...          mu       = .1,
...          stepSize = 1e-2,
...          BCs      = 'per2',
...          numPlots = 100,
...          timeEnd  = .1
...          )
>>> path.evolve() #doctest: +ELLIPSIS
<__main__.Path object at 0x...>
>>> len(path.evolution)
11
```

```
    """

    def __init__(self,
                 initial   = None,
                 eigval    = None,
                 solver    = 'per2',
                 evolution = None,
                 solved    = False,
                 norms     = None,
                 energies  = None,
                 BCs       = 'per2',
                 lambd     = 500.0,
                 sigma     = 0.0,
                 mu        = 0.0,
                 stepSize  = 1e-5,
                 timeEnd   = 0.1,
                 numPlots  = 100,
                 cache     = None):

        self._initial   = initial
        self._lambda    = lambd
        self._sigma     = sigma
        self._mu        = mu
        self._eigval    = eigval
        self._solver    = solver
```

```
        self._solved     = solved
        self._BCs        = BCs
        self._stepSize   = stepSize
        self._timeEnd    = timeEnd
        self._numPlots   = numPlots
        self._cache      = cache
        self._times      = None

        if norms == None:
            self._norms = []
        else:
            self._norms = norms

        if energies == None:
            self._energies = []
        else:
            self._energies = energies

        if evolution == None:
            self._evolution = []
        else:
            self._evolution = evolution

    @property
    def initial(self):
```

```
        """The initial condition"""

        if self._initial != None:
            return self._initial.copy()
        else: return None

    @property
    def eigval(self):
        """The eigenvalue which corresponds to this path."""

        return self._eigval

    @property
    def solver(self):
        """The path to the timestepper, solver."""

        return self._solver

    @property
    def evolution(self):
        """The evolution as calculated by solver"""

        return copy.copy(self._evolution)

    @property
```

```
    def solved(self):
        """True if the evolution has been calculated."""

        return self._solved

    @solved.setter
    def solved(self,value):
        self._solved = value

    @property
    def norms(self):
        """A list of the norms of the fourier coefficients as computed by
        solver."""

        return copy.copy(self._norms)

    @property
    def energies(self):
        """A list of the energies computed by the solver ten times per plot
        step."""
        return copy.copy(self._energies)

    @property
```

```
    def times(self):
        """A list of times at which the evolutions and energies were
        recorded."""

        return copy.copy(self._times)

    @property
    def BCs(self):
        """The Boundary conditions. Available choices:
            'neu1'
            'neu2'
            'neu3'
            'per1'
            'per2'
            'per3'
            'dir1'
            'dir2'"""
        return self._BCs

    @property
    def lambd(self):
        """lambda"""
        return self._lambda

```

```
    @property
    def sigma(self):
        """sigma"""
        return self._sigma

    @property
    def mu(self):
        """mu"""
        return self._mu

    @property
    def timeEnd(self):
        """End time of simulation. This will be shifted if continuing a run so
        that the entire simulation time is timeEnd."""
        return self._timeEnd

    @timeEnd.setter
    def timeEnd(self,value):
        self._timeEnd = value

    @property
    def numPlots(self):
        """The number of plots to be generated during this evolution. If
```

```
        continuing a run, this number of plots will be generated for the
        continuation."""
        return self._numPlots

    @numPlots.setter
    def numPlots(self, value):
        self._numPlots = value

    @property
    def cache(self):
        """The location of the cache directory for the evolution. If set to
        none, the evolution is not cached. An absolute path is necessary if you
        plan on opening the same evolution from different working directories."""
        return self._cache

    @cache.setter
    def cache(self, value):
        self._cache = value

    @property
    def stepSize(self):
```

```
        """The stepSize is the size of the time step used by the solver. The
        default, 1e-5 should give stability for most conditions on the DBCP
        model."""
        return self._stepSize

    @stepSize.setter
    def stepSize(self, value):
        self._stepSize = value

    def evolve(self):
        """Uses solver to evolve. The path is returned after evolution. If
        solved, the last snapshot of the evolution is used as the initial
        condition."""

        if self.cache != None and not os.path.exists(self.cache):
            os.mkdir(self._cache)

        assert not (self.evolution == [] and self.solved)

        if self.solved:
            if self.cache != None:
```

```
                cacheLoc  = len(self.evolution)
                snap      = _loadCache(cacheLoc-1, self.cache)
                initial   = snap.soln
                timeShift = snap.time
            else:
                initial = self.evolution[-1].soln
                timeShift = self.evolution[-1].time
        else:
            initial = self._initial
            self._evolution = []
            timeShift = 0
            cacheLoc = 0

        assert (self.timeEnd - timeShift) > 0
        if self.timeEnd - timeShift == 0:
            return self

        process = subprocess.Popen([self._solver],
                                   shell=False,
                                   stdin=subprocess.PIPE,
                                   stdout=subprocess.PIPE)

        stdin = str(self.mu) + ' '\
              + str(self.lambd) + ' '\
              + str(self.sigma) + ' '\
```

```
            + str(self.stepSize) + ' '\
            + str(self.timeEnd - timeShift) + ' '\
            + str(self.numPlots)

        if initial != None:
            stdin += '\n1\n'
            for element in initial.flatten():
                stdin += '%g %g\n' % (element.real, element.imag)
        else:
            stdin += '\n0';

        message = process.communicate(stdin)[0]
        lines = message.split('\n')

        for line in lines:

            line = line.lower()
            data = line.split()

            # There needs to be a meaning when these are evaled
            nan = numpy.nan
            inf = numpy.infty
            nannanj = numpy.nan

            if len(data) == 0:
```

```
                pass
            elif data[0] == 'norm':
                self._norms.append(eval(data[1]))

            elif data[0] == 'begin' and data[1] == 'plot':
                snap = []

            elif data[0] == 'time':
                time = eval(data[1]) + timeShift

            elif data[0] == 'end' and data[1] == 'plot':
                snap = numpy.array(snap)
                snapshot = Snapshot(time,snap)
                if self.cache != None:
                    _saveCache(snapshot, cacheLoc, self.cache)
                    self._evolution.append(cacheLoc)
                    cacheLoc += 1
                else:
                    self._evolution.append(snapshot)

            else:
                snap.append(eval(line))

        self.solved = True

```

```
        return self

    def plot(self, filepath):
        """Plots each snapshot using matplotlib for 1D and 2D plots and vtk for
        3D plots.in the given directory."""

        if not os.path.exists(filepath):
            os.mkdir(filepath)
        else:
            print >> sys.stderr, "Warning: Plot directories exist"

        if len(self.evolution) > 0:
            if self.cache == None:
                dim = len(self.evolution[0].soln.shape)
            else:
                snap = _loadCache(self.evolution[0], self.cache)
                dim  = len(snap.soln.shape)
            if dim == 1:
                plotter = plotters.plot
            if dim == 2:
                plotter = plotters.pcolor
            else:
                plotter = plotters.plotVTK
```

```
        for snapshot in self.evolution:
            if self.cache != None:
                snapshot = _loadCache(snapshot, self.cache)

            soln = snapshot.soln
            time = snapshot.time
            u = prePlot(soln, self.BCs)
            plotter(u,filepath + '/%.08f' % time)

    def computeEnergies(self):
        """Compute the energy for each snapshot in evolution. Currently, this
        doesn't work for certain boundaries or dimensions!."""

        if not self.solved:
            return
        else:
            self._energies = []
            self._times = []
            if self.BCs[:3] == 'per':
                if self.cache != None:
                    firstPlot = _loadCache(self.evolution[0], self.cache)
                else:
```

```
                    firstPlot = self.evolution[0]
                shape = firstPlot.soln.shape
                kappa = _genEigenvalues(shape, self.BCs)

                for snap in self.evolution:

#FIXME none of this works for 1D due to the choice for N
#FIXME none of this works for neumann boundaries
                    if self.cache != None:
                        snap = _loadCache(snap, self.cache)

                    a = snap.soln
                    d = a.ndim
                    N = a.shape[0]

                    u = prePlot(a,
                                BCs=self.BCs,
                                resolution=3*N
                                )
                    w = (1 - u**2)**2 / 4.
                    b = BC_funcs[self.BCs](w) / N**d

                    energy = 0
                    coords = [range(i) for i in shape]
                    coords = product(*coords)
```

```
                    for coord in coords:
                        if any(coord) == 0:
                            energy += b[coord]
                        elif coord[-1] == shape[-1]-1:
                            energy += a[coord] * numpy.conj(a[coord])\
                                *(kappa[coord] / self.lambd\
                                + self.sigma * .5 / kappa[coord])
                        else:
                            energy += 2 * a[coord] * numpy.conj(a[coord])\
                                *(kappa[coord] / self.lambd\
                                + self.sigma * .5 / kappa[coord])

                    self._energies.append(energy)
                    self._times.append(snap.time)
            else:
                raise NotImplementedError



    def save(self, filename):
        """Save this Path to a file."""
```

```
        outfile = gzip.open(filename + '.gz', 'w')
        cPickle.dump(self, outfile)
        outfile.close()

class Snapshot(object):
    """This class will contain two attributes. Namely, time and solution. They
    correspond to the time at which solver has returned values for a given
    evolution"""

    def __init__(self, time, soln):
        self._time = time
        self._soln = soln

    @property
    def time(self):
        """The time at which solver computed this solution snapshot"""
        return self._time

    @property
    def soln(self):
        """The solution which solver computed for this snapshot"""
```

```
        return self._soln.copy()

def prePlot(a, BCs='neu', resolution=256):
    """ Computes the inverse transform of an array of fourier coefficients, a,
    after padding the array so that the array length is at least the specified
    resolution. This padded array is then returned.

    >>> [x,y] = numpy.mgrid[0:1:1./64,0:1:1./64]
    >>> u     = numpy.sin(2*pi*(x+y))
    >>> a     = BC_funcs['per2'](u) / 64**2
    >>> u0    = prePlot(a,'per2',64)
    >>> u1    = prePlot(a,'per2',128)

    u should be close to the ifft(fft(u))
    >>> numpy.allclose(u,u0)
    True

    max/min should stay the same regardless of resolution
    >>> numpy.allclose([u1.max(),u1.min()],[u.max(),u.min()])
    True

    The same should be true in 3D.

```

```
    >>> [x,y,z] = numpy.mgrid[0:1:1./64,0:1:1./64,0:1:1./64]
    >>> u       = numpy.sin(2*pi*(x+y+z))
    >>> a       = BC_funcs['per3'](u) / 64**3
    >>> u0      = prePlot(a,'per3',64)
    >>> u1      = prePlot(a,'per3',128)

    u should be close to the ifft(fft(u))
    >>> numpy.allclose(u,u0)
    True

    max/min should stay the same regardless of resolution
    >>> numpy.allclose([u1.max(),u1.min()],[u.max(),u.min()])
    True

    It would be nice if the default resolution does not throw a
        MemoryError.
    >>> u2 = prePlot(a,'per3')
    """
    a = a.copy()

    rt = BC_ifuncs[BCs]

    for size in a.shape:
        if resolution < size:
            resolution = size
```

```
            print >> sys.stderr, "Warning: array size used instead of"\
                                    +"specified resolution."

    if len(a.shape) == 1:
        size = a.shape[0]

        if BCs == 'per':
            resolution = resolution/2 + 1

        padsize = resolution-size

        padding = numpy.zeros(padsize)
        aext = numpy.concatenate((a,padding))

        aext *= sqrt(resolution)

    elif len(a.shape) == 2:

        if BCs == 'per2':
            if a.shape[0] == resolution:
                aext = a
            else:
                n2 = a.shape[0]/2
                nh = a.shape[1]-1
```

```
                aext = numpy.zeros([resolution,resolution/2 + 1]
                                    ,numpy.complex128)
                aext[:n2,:nh] = a[:n2,:nh]
                aext[:n2,nh]  = .5 * a[:n2,nh]
                aext[-n2+1:,:nh] = a[-n2+1:,:nh]
                aext[-n2+1:,nh] = .5 * a[-n2+1:,nh]

                if a.shape[0] % 2 == 0:
                    aext[-n2,:nh] = .5 * a[n2,:nh]
                    aext[n2,:nh]  = .5 * a[n2,:nh]
                    aext[-n2,nh]  = .25 * a[n2,nh]
                    aext[n2,nh]   = .25 * a[n2,nh]

            aext *= resolution**2

        else:
            aext = numpy.zeros([resolution,resolution])
            aext[:a.shape[0],:a.shape[1]] = a
            aext *= resolution

    elif len(a.shape) == 3:
        if BCs == 'per3':
            assert a.shape[0] == a.shape[1]
            if a.shape[0] == resolution:
```

```
            aext = a
        else:
            n2 = a.shape[0]/2
            nh = a.shape[-1]-1

            aext = numpy.zeros([resolution,resolution,
                resolution/2 +1]
                                ,numpy.complex128)

            aext[:n2,:n2,:nh]          = a[:n2,:n2,:nh]
            aext[-n2+1:,:n2,:nh]       = a[-n2+1:,:n2,:nh]
            aext[:n2,-n2+1:,:nh]       = a[:n2,-n2+1:,:nh]
            aext[-n2+1:,-n2+1:,:nh] = a[-n2+1:,-n2+1:,:nh]

            aext[:n2,:n2,nh]           = .5 * a[:n2,:n2,nh]
            aext[-n2+1:,:n2,nh]        = .5 * a[-n2+1:,:n2,nh]
            aext[:n2,-n2+1:,nh]        = .5 * a[:n2,-n2+1:,nh]
            aext[-n2+1:,-n2+1:,nh] = .5 * a[-n2+1:,-n2+1:,nh]

            if a.shape[0] %2 == 0:
                aext[-n2,:n2,:nh] = .5 * a[n2,:n2,:nh]
                aext[n2,:n2,:nh]  = .5 * a[n2,:n2,:nh]
                aext[:n2,-n2,:nh] = .5 * a[:n2,n2,:nh]
                aext[:n2,n2,:nh]  = .5 * a[:n2,n2,:nh]

```

```
aext[-n2,-n2+1:,:nh] = .5 * a[n2,-n2+1:,:nh]
aext[n2,-n2+1:,:nh]  = .5 * a[n2,-n2+1:,:nh]
aext[-n2+1:,-n2,:nh] = .5 * a[-n2+1:,n2,:nh]
aext[-n2+1:,n2,:nh]  = .5 * a[-n2+1:,n2,:nh]

aext[-n2,:n2,nh] = .25 * a[n2,:n2,nh]
aext[n2,:n2,nh]  = .25 * a[n2,:n2,nh]
aext[:n2,-n2,nh] = .25 * a[:n2,n2,nh]
aext[:n2,n2,nh]  = .25 * a[:n2,n2,nh]

aext[-n2,-n2+1:,nh] = .25 * a[n2,-n2+1:,nh]
aext[n2,-n2+1:,nh]  = .25 * a[n2,-n2+1:,nh]
aext[-n2+1:,-n2,nh] = .25 * a[-n2+1:,n2,nh]
aext[-n2+1:,n2,nh]  = .25 * a[-n2+1:,n2,nh]

aext[-n2,n2,:nh]  = .25 * a[n2,n2,:nh]
aext[n2,n2,:nh]   = .25 * a[n2,n2,:nh]
aext[n2,-n2,:nh]  = .25 * a[n2,n2,:nh]
aext[-n2,-n2,:nh] = .25 * a[n2,n2,:nh]

aext[-n2,n2,nh]  = .125 * a[n2,n2,nh]
aext[n2,n2,nh]   = .125 * a[n2,n2,nh]
aext[n2,-n2,nh]  = .125 * a[n2,n2,nh]
aext[-n2,-n2,nh] = .125 * a[n2,n2,nh]
```

```
773                 aext *= resolution**3
774
775             else:
776                 aext = numpy.zeros([resolution,resolution,resolution
                        ])
777                 aext[:a.shape[0],:a.shape[1],:a.shape[2]] = a
778                 aext *= resolution**(1.5)
779         else:
780             raise NotImplementedError
781
782         u = rt(aext)
783
784         return u
785
786 def runBuilder(conditions, baseDir='.', ncpus=None):
787         """This function takes a list of lists of the form
788             [paramdict, number] or [paramdict, Path, dirname],
789         where paramdict is a dictionary of parameters for the
                creation of a path
790         object and number is the number of runs to perform with these
                parameters.
791         After the path objets have been created, they are evolved,
                pickled and
792         saved within the baseDir in parallel with ncpus workers.
793
```

```
    The second format is for continuing a run. Path is a Path object, dirname
    is where you'd like the plots and pickle stored, and paramdict is the same
    as before, but can only contain attributes in the public interface of Path
    (solved, numPlots and 'timeEnd').

    The following raises an AssertionError since timeEnd is set to 0.
    Otherwise, it is a good example.

    >>> conditions = [[{ 'solver'  : 'per2',
    ...                  'BCs'     : 'per2',
    ...                  'lambd'   : 500.0,
    ...                  'sigma'   : 0.0,
    ...                  'mu'      : 0.0,
    ...                  'numPlots': 100,
    ...                  'timeEnd' : 0.0},
    ...                  1]]
    >>> try:
    ...     runBuilder(conditions)
    ... except AssertionError:
    ...     pass
    """
```

```
    paths = []
    for condition in conditions:
        if type(condition[1]) == int:
            for i in range(condition[1]):
                path = (Path(**condition[0]),
                        i)
                paths.append(path)
        else:
            for key in condition[0]:
                if key == 'numPlots':
                    condition[1].numPlots = condition[0][key]
                elif key == 'timeEnd':
                    condition[1].timeEnd = condition[0][key]
                elif key == 'solved':
                    condition[1].solved = condition[0][key]
                elif key == 'stepSize':
                    condition[1].stepSize = condition[0][key]
                else:
                    raise Exception('Invalid paramdict for run restart')
            paths.append((condition[1],condition[2]))

    if ncpus == None:
        ncpus = multiprocessing.cpu_count()
```

```
    pool   = multiprocessing.Pool(processes=ncpus)
    runner = functools.partial(_runner, dirname=baseDir)

    pool.map(runner,paths)

def _runner(path, dirname):
    i     = path[1]
    path  = path[0]
    sigma = path.sigma
    lambd = path.lambd
    mu    = path.mu

    if dirname[-1] != '/':
        dirname += '/'

    if type(i) == str:
        dirname += i
    else:
        dirname += str(mu) + '_'\
                 + str(sigma) + '_'\
                 + str(lambd) + '/'\
                 + str(i) + '/'

    if not os.path.exists(dirname):
```

```
        os.makedirs(dirname)

    if path.cache == '':
        path.cache = dirname + 'evocache'
    path.evolve()
    path.save(dirname + 'evo')
    path.plot(dirname)

def _loadCache(cacheLoc,cache):
    loadFile = open(cache + '/' + str(cacheLoc), 'r')
    snapshot = cPickle.load(loadFile)
    loadFile.close()
    return snapshot

def _saveCache(obj, cacheLoc, cache):
    dumpFile = open(cache + '/' + str(cacheLoc), 'w')
    cPickle.dump(obj, dumpFile)
    dumpFile.close()

def _genEigenvalues(shape,BCs):
    N = shape[0]
    if BCs[:3] == 'per':
        kappap = [4 * pi**2 * i**2 for i in range(N/2) + range(N
            /2,0,-1)]
        kappa  = numpy.zeros(shape)
```

```
        coords = [range(i) for i in shape]
        coords = product(*coords)
        for coord in coords:
            eigval = [kappap[i] for i in coord]
            kappa[coord] = sum(eigval)
        return kappa
    else:
        raise NotImplementedError

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

B.0.2 plotters.py

```
import numpy, sys
import matplotlib.pyplot as pyplt
from pyvtk import VtkData, StructuredPoints, PointData, Scalars

def plot(u, filename):
    """ Plots the given array and saves it to the given filename.
        """

    N = u.shape[0]
    x = numpy.linspace(0,1,N)

    pyplt.figure()
```

```
    pyplt.plot(x,u)
    pyplt.savefig(filename + '.png')
    pyplt.close()

def pcolor(u, filename):
    """ Plots the given array and saves it to the given filename.
        """

    if u.max() > 1 or u.min() <-1:
        print >> sys.stderr, "plot out of bounds"

    pyplt.figure()
    pyplt.axes([0.0, 0.0, 1.0, 1.0])
    try:
        pyplt.imsave(arr=u,fname=filename + '.png',origin='lower'
            ,vmin=-1,vmax=1)
    except AttributeError:
    # Older versions of matplotlib do not have imsave, so it is
        implemented
    # below.
        from matplotlib.backends.backend_agg import
            FigureCanvasAgg
        from matplotlib.figure import Figure

        fig = Figure(figsize=u.shape[::-1],dpi=1,frameon=False)
```

```
        canvas = FigureCanvasAgg(fig)
        fig.figimage(u,vmin=-1,vmax=1,origin='lower')
        fig.savefig(filename + '.png',dpi=1)

    pyplt.close()

def plotVTK(u, filename):
    """Saves a given array in a vtk file with the given filename.
        """

    shape = u.shape
    VtkData(StructuredPoints(shape),
            PointData(Scalars(u.flatten()))
            ).tofile(filename + '.vtk')
```

B.0.3 transFunc.py

```
import numpy
from math import sqrt

#TODO add doctests for this module

def matlab_dct(v,axis=-1):
    """Implements the Matlab version of the dct, i.e., the
        nonscaled version of
    the dct implemented in dct is scaled to normalize it and make
        it
```

```
    orthogonal."""

    N = v.shape[axis]

    vhat = dct(v,axis)

    if axis == 0:
        vhat[0,...] = vhat[0,...] * 1.0/(2.0 * sqrt(N))
        vhat[1:,...] = vhat[1:,...] * 1.0/sqrt(2.0 * N)
    elif axis == -1:
        vhat[...,0] = vhat[...,0] * 1.0/(2.0 * sqrt(N))
        vhat[...,1:] = vhat[...,1:] * 1.0/sqrt(2.0 * N)
    elif axis == -2:
        vhat[...,0,:] = vhat[...,0,:] * 1.0/(2.0 * sqrt(N))
        vhat[...,1:,:] = vhat[...,1:,:] * 1.0/sqrt(2.0 * N)

    return vhat


def dct(v,axis=-1):
    """Implements the discrete cosine transform by computing a fourier
    transform of an array of length 4N."""

    N = v.shape[axis]
```

```
    vp = numpy.zeros(4*N)

    slices = [None]*2
    slices[0] = slice(1,2*N+1,2)
    slices[1] = slice(4*N+1,2*N,-2)

    vp[slices[0]] = v
    vp[slices[1]] = v

    vhat = numpy.fft.rfft(vp)

    return vhat[:N]

def dct2(v,axes=(-1,0)):
    """Implements Matlab's dct2 routine."""

    return matlab_dct(matlab_dct(v,axis=axes[0]),axis=axes[1])

def dct3(v,axes=(-1,-2,0)):
    """Implements Matlab's dct3 routine."""

    return matlab_dct(matlab_dct(matlab_dct(v
                        ,axis=axes[0])
                        ,axis=axes[1])
                        ,axis=axes[2])
```

```
def matlab_idct(v,axis=-1):
    """
    Implememnts the Matlab version of idct, i.e., it uses
    the basis scalings as in Matlab's dct/idct pair.
    """

    N = v.shape[axis]
    v=v.copy()
    if axis == 0:
        v[0,...] = v[0,...] * 2.0*sqrt(1.0/N)
        v[1:,...] = v[1:,...] * sqrt(2.0/N)
    elif axis == -2:
        v[...,0,:] = v[...,0,:] * 2.0*sqrt(1.0/N)
        v[...,1:,:] = v[...,1:,:] * sqrt(2.0/N)
    elif axis == -1:
        v[...,0] = v[...,0] * 2.0*sqrt(1.0/N)
        v[...,1:] = v[...,1:] * sqrt(2.0/N)
    return idct(v,axis) / (2.0 / N)

def idct(v,axis=-1):
    """
    Implements the inverse discrete cosine transform,
    with a slightly nonstandard scaling.
    """
```

```
    pi = numpy.pi
    n = len(v.shape)
    N = v.shape[axis]
    even = (N%2 == 0)
    slices = [None]*4
    for k in range(4):
        slices[k] = []
        for j in range(n):
            slices[k].append(slice(None))
    k = numpy.arange(N)
    if even:
        ak = numpy.r_[1.0,[2]*(N-1)]*numpy.exp(1j*pi*k/(2*N))
        newshape = numpy.ones(n)
        newshape[axis] = N
        ak.shape = newshape
        xhat = numpy.real(numpy.fft.ifft(v*ak,axis=axis))
        x = 0.0*v
        slices[0][axis] = slice(None,None,2)
        slices[1][axis] = slice(None,N/2)
        slices[2][axis] = slice(N,None,-2)
        slices[3][axis] = slice(N/2,None)
        for k in range(4):
            slices[k] = tuple(slices[k])
        x[slices[0]] = xhat[slices[1]]
        x[slices[2]] = xhat[slices[3]]
```

```
        return x
    else:
        ak = 2*numpy.exp(1j*pi*k/(2*N))
        newshape = numpy.ones(n)
        newshape[axis] = N
        ak.shape = newshape
        newshape = list(v.shape)
        newshape[axis] = 2*N
        Y = numpy.zeros(newshape,numpy.complex)
        #Y[:N] = ak*v
        #Y[(N+1):] = conj(Y[N:0:-1])
        slices[0][axis] = slice(None,N)
        slices[1][axis] = slice(None,None)
        slices[2][axis] = slice(N+1,None)
        slices[3][axis] = slice((N-1),0,-1)
        Y[slices[0]] = ak*v
        Y[slices[2]] = numpy.conj(Y[slices[3]])
        x = numpy.real(numpy.fft.ifft(Y,axis=axis))[slices[0]]
        return x

def idct2(v,axes=(-1,-2)):
    """
    Matlab's idct2 routine.
    """
    return matlab_idct(matlab_idct(v,axis=axes[0]),axis=axes[1])
```

```
def idct3(v,axes=(-1,-2,0)):
    """Implements Matlab's idct3 routine."""

    return matlab_idct(matlab_idct(matlab_idct(v
                        ,axis=axes[0])
                        ,axis=axes[1])
                        ,axis=axes[2])

def matlab_dst(v,axis=-1):
    """Matlab's dst routine."""
    #TODO implement dst
    raise NotImplementedError()


def matlab_idst(v,axis=-1):
    """Matlab's idst routine."""
    #TODO implement idst
    raise NotImplementedError()

def dst2(v,axes=(-1,-2)):
    """Matlab's dst2 routine."""
    #TODO implement dst2
    raise NotImplementedError()

```

```
def idst2(v,axes=(-1,-2)):
    """Matlab's idst2 routine."""
    #TODO implement idst2
    raise NotImplementedError()
```

B.1 Examples

B.1.1 CMWcheck.py

```
#!/Users/matkins/bin/python
import numpy, util

# Number of cores to use
ncpus = 3

# Number of parameter possibilities this translates to N * 4 runs
    , so be careful
N = 3

# Amount of time to run for.
time = 100000

# Amount of perturbation away from the asymptotic regime boundary
perturbation = .001

# Domain sizes
L = [4 * numpy.pi,
```

```
        2 * numpy.pi,
        numpy.pi
      ]

# Base directory to store the plots in
baseDir = '/Users/matkins/scratch/CMWCheck'

mu1 = numpy.random.uniform(0,.2,N)
mu2 = numpy.random.uniform(0,.6,N)

# Use the below function to get gamma from beta and mu
f = lambda beta, mu : (2./\
                        (1 - 3*(mu/beta)**2),
                        mu)

# Use the below function to get mu from beta and gamma
#f = lambda beta, gamma : beta * numpy.sqrt(1./3. - 2./(3.*gamma))

# Lamellae condition is beta less than the following
beta1 = (1./29.) * numpy.sqrt(551-174*numpy.sqrt(6))

# Hex-circ condition is beta less than the following
beta2 = 3*numpy.sqrt(5./37.)

```

```
gammal = [f(beta1 - perturbation, m) for m in mu1]
print gammal
gammac = [f(beta1 + perturbation, m) for m in mu1]\
        + [f(beta2 - perturbation, m) for m in mu2]
print gammac
gammad = [f(beta2 + perturbation, m) for m in mu2]

# In the CMW paper, sigma = 1
sigma = 1

def domSz(gamma):
    if gamma < 10:
        return L[0]
    elif gamma < 25:
        return L[1]
    else:
        return L[2]

parmsl = [(g**2*domSz(g)**2,
           m,
           sigma*domSz(g)**2,
           (.1/(1+g**1.5))/domSz(g)**4/g**2,
           time/domSz(g)**4/g**2)
           for (g,m) in gammal]

```

```
parmsc = [(g**2*domSz(g)**2,
          m,
          sigma*domSz(g)**2,
          (.1/(1+g**1.5))/domSz(g)**4/g**2,
          time/domSz(g)**4/g**2)
          for (g,m) in gammac]

parmsd = [(g**2*domSz(g)**2,
          m,
          sigma*domSz(g)**2,
          (.1/(1+g**1.5))/domSz(g)**4/g**2,
          time/domSz(g)**4/g**2)
          for (g,m) in gammad]

mkrun = lambda parm: ({'solver'   : 'per2',
                       'BCs'      : 'per2',
                       'lambd'    : parm[0],
                       'sigma'    : parm[2],
                       'mu'       : parm[1],
                       'numPlots': 100,
                       'timeEnd' : parm[4],
                       'stepSize': parm[3]
                       },
                       1)
```

```
lamellaeRuns = [mkrun(parm) for parm in parmsl]
util.runBuilder(lamellaeRuns, baseDir+'/lamellae', ncpus)

circularRuns = [mkrun(parm) for parm in parmsc]
util.runBuilder(circularRuns, baseDir+'/circular', ncpus)

disorderRuns = [mkrun(parm) for parm in parmsd]
util.runBuilder(disorderRuns, baseDir+'/disorder', ncpus)
```

B.1.2 dg.py

```
#!/Users/matkins/bin/python

import util

epsilon = .03
mu = .2
lambd = 2 / epsilon**2
sigma = epsilon * 2**11 * (1 - mu**2)**-2

conditions = [[{ 'mu'        : mu,
                 'lambd'     : lambd,
                 'sigma'     : sigma,
                 'BCs'       : 'per3',
                 'solver'    : 'per3',
                 'numPlots'  : 10,
                 'timeEnd'   : 1,
```

```
                    'cache'     : '',
                    'stepSize' : 1e-5},
                    4]]

util.runBuilder(conditions, baseDir='/Users/matkins/scratch/raid/
    dgc')
```

Appendix C: Solvers

C.1 2D_per.c

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <fftw3.h>
#include <time.h>
#include "mt.h"

#define DEBUG    (0)

#define NORMOUT (1)

#define N        (40)
#define NH       ((N) / 2 + 1)
#define NLONG    (3 * N)
#define NLONGH   ((NLONG) / 2 + 1)

#define PI       (3.141592653589793)

#define sq(x)    ((x)*(x))
#define cu(x)    ((x)*sq(x))

#define coordh(i,j) ((j) + NH * (i))
```

```
23 #define coordlongh(i,j) ((j) + NLONGH * (i))
24 #define coordlong(i,j) ((j) + NLONG * (i))
25
26 void scale(fftw_complex *b, fftw_complex *bext){
27     //Copys data from the array the transform was calculated on
           and scales the
28     //data simultanaeously.
29
30     register int i,j;
31
32     for (i=0;i<N/2;i++){
33         for (j=0;j<NH-1;j++){
34             b[coordh(i,j)] = bext[coordlongh(i,j)] / sq(NLONG);
35             if (i != 0) {
36                 b[coordh(N-i,j)] = bext[coordlongh(NLONG-i,j)] /
                       sq(NLONG);
37             }
38         }
39     }
40 }
41
42 void print_complex(fftw_complex *a, int n, int nh){
43     //Prints an n x nh array of complex numbers to stdout in a
           way that
44     //util will be able to read
```

```
45      register int i,j;
46
47      for (i=0;i<n;i++) {
48          printf("[");
49          for (j=0;j<nh;j++) {
50              switch (nh) {
51                  case NH:
52                      printf("%lg%+lgj",
53                              creal(a[coordh(i,j)]),
54                              cimag(a[coordh(i,j)]));
55                      break;
56                  case NLONGH:
57                      printf("%lg%+lgj",
58                              creal(a[coordlongh(i,j)]),
59                              cimag(a[coordlongh(i,j)]));
60                      break;
61                  }
62              if (j < nh-1) {
63                  printf(",");
64              }
65          }
66          printf("]\n");
67      }
68  }
69
```

```
70 void print_double(double *a, int n){
71     //Prints an n x n array of double to stdout in a way that
           util can
72     //decipher.
73     register int i,j;
74
75     for (i=0;i<n;i++) {
76         printf("[");
77         for (j=0;j<n;j++) {
78             printf("%lg", a[coordlongh(i,j)]);
79             if (j < n-1) {
80                 printf(",");
81             }
82         }
83         printf("]\n");
84     }
85 }
86
87 double norm(fftw_complex *a){
88     //computes the two-norm in fourier space
89
90     register int i;
91     double value;
92
93     value = 0.0;
```

```
    for (i=0;i<N*NH;i++){
        value += (double)(a[i] * conj(a[i]));
    }
    return sqrt(value);
}

void b_update(double *u, double *f, fftw_plan fft, fftw_plan ifft
    ){
    //computes the non-linearity by computing the ifft, computing
        the
    //non-linearity and computing the fft of that.

    register int i;

    fftw_execute(ifft);

    for (i=0;i<sq(NLONG);i++){
        f[i] = u[i] - cu(u[i]);
    }

    fftw_execute(fft);
}

void a_copy(fftw_complex *a, fftw_complex *aext){

```

```
117     register int i,j;
118
119     // Zero the array (FFTW will destroy it, so it needs to be
            rezeroed at
120     // everytime step).
121     for (i=0;i<NLONG*NLONGH;i++){
122         aext[i] = 0;
123     }
124
125     // Copy the smaller array into the padded one so that
            symmetry is
126     // preserved.
127     for (i=0;i<N/2;i++){
128         for (j=0;j<NH-1;j++) {
129             aext[coordlongh(i,j)] = a[coordh(i,j)];
130             if (i != 0) {
131                 aext[coordlongh(NLONG-i,j)] = a[coordh(N-i,j)];
132             }
133         }
134     }
135 }
136
137 main(){
138
139     register int i,j,k;
```

```
140     int initial, numplots, plotstep, normstep;
141     double *u, *f, *kappa, *kappap, mu, sigma, lambda, tmp_real,
            tmp_imag,
142             h, timeend;
143     fftw_complex *a, *b, *aext, *bext;
144     fftw_plan fft, ifft;
145
146     kappa = (double*) fftw_malloc(sizeof(double) * N * NH);
147     kappap= (double*) fftw_malloc(sizeof(double) * N);
148     a     = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N
            * NH);
149     b     = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N
            * NH);
150     aext  = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) *
            NLONG * NLONGH);
151     bext  = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) *
            NLONG * NLONGH);
152     u     = (double*) fftw_malloc(sizeof(double) * sq(NLONG));
153     f     = (double*) fftw_malloc(sizeof(double) * sq(NLONG));
154
155     fftw_import_system_wisdom();
156     fft   = fftw_plan_dft_r2c_2d(NLONG,NLONG, f, bext,
            FFTW_EXHAUSTIVE);
157     ifft  = fftw_plan_dft_c2r_2d(NLONG,NLONG, aext, u,
            FFTW_EXHAUSTIVE);
```

```
158
159     // Read coefficients and other parameters from stdin
160     scanf("%lf %lf %lf %lf %lf %d",
161           &mu, &lambda, &sigma, &h, &timeend, &numplots);
162     plotstep = (int) ceil(timeend / h / numplots);
163     normstep = (int) ceil(timeend / h / (numplots*10));
164
165
166     // See if stdin tells us to use an initial condition
167     // 0 means no, 1 means yes
168     scanf("%d", &initial);
169
170     // Initializtion of eigenvalue array
171
172     for (i=0;i<N/2;i++) {
173         kappap[i] =  4.0 * sq(i) * sq(PI);
174         kappap[i+N/2] =  4.0 * sq(N/2-i) * sq(PI);
175     }
176
177     for (i=0;i<N;i++) {
178         for (j=0;j<NH;j++) {
179             kappa[coordh(i,j)] = kappap[i] + kappap[j];
180         }
181     }
182
```

```
183     if (initial) {
184         for (i=0;i<N*NH;i++) {
185             scanf("%lf %lf", &tmp_real, &tmp_imag);
186             a[i] = tmp_real + I*tmp_imag;
187         }
188     }
189
190     // Use random initial conditions otherwise
191     else {
192         // Initialization of the coefficient array
193         // (first we need a seed)
194         init_genrand(time(NULL));
195
196         // Random coefficients in real space
197         for (i=0;i<NLONG;i++) {
198             for (j=0;j<NLONG;j++) {
199                 f[coordlong(i,j)] = genrand_realc();
200             }
201         }
202
203         fftw_execute(fft);
204
205         // Zero a before copy
206         for (i=0;i<N*NH;i++) {
207             a[i] = 0.0;
```

```
208             }
209
210             scale(a,bext);
211             a[0] = mu;
212         }
213
214         // Initialize b to zeros
215         for (i=0;i<N*NH;i++) {
216             b[i] = 0.0;
217         }
218
219 #if DEBUG
220         printf("a\n");
221         print_complex(a,N,NH);
222
223         a_copy(a,aext);
224
225         printf("\naext\n");
226         print_complex(aext,NLONG,NLONGH);
227
228         fftw_execute(ifft);
229
230         //print_double(u,NLONG);
231
232         for (i=0;i<cu(NLONG);i++){
```

```
233             f[i] = u[i];
234         }
235
236         //print_double(f,NLONG);
237
238         fftw_execute(fft);
239
240         printf("\nbext\n");
241         print_complex(bext,NLONG,NLONGH);
242
243         scale(b,bext);
244
245         printf("\nb\n");print_complex(b,N,NH);
246
247         int works = 1;
248         for (i=0;i<N;i++) {
249             for (j=0;j<NH;j++) {
250                 if ((double) ((a[coordh(i,j)]-b[coordh(i,j)])
251                                 * conj(a[coordh(i,j)]-b[coordh(i,j)]))
252                                 > 1e-16) {
253                     works = 0;
254                     printf("\nTransform not invertible, at (%d,%d),\
255                         %lf%+lfi != %lf%+lfi\n",
256                                 i,j,
```

```
                        creal(a[coordh(i,j)]),cimag(a[coordh(i,j)]),
                        creal(b[coordh(i,j)]),cimag(b[coordh(i,j)]));
            }
        }
    }
    if (works) {
        printf("\nTransform is invertible!\n");
    }

#else
    for (k=0;k*h<timeend;k++) {

        if(k % plotstep == 0) {
            printf("begin plot\n");
            printf("time %lg\n", k*h);
            print_complex(a,N,NH);
            printf("end plot\n");
        }
#if NORMOUT
        if(k % normstep == 0) {
            printf("norm %lg\n",norm(a));
        }
#endif
```

```
        a_copy(a,aext);
        b_update(u,f,fft,ifft);
        scale(b,bext);

        for (i=0;i<N;i++) {
            for (j=0;j<NH-1;j++) {
                if (!(i == N/2 && N%2 == 0) && !(i == 0 && j == 0)) {
                    a[coordh(i,j)] = (a[coordh(i,j)]
                        + h * lambda * b[coordh(i,j)] * kappa[coordh(i,j)])
                        / (1 + h * sq(kappa[coordh(i,j)]) + h * lambda * sigma);
                }
            }
        }
    }
    printf("begin plot\n");
    printf("time %lf\n", timeend);
    print_complex(a,N,NH);
    printf("end plot\n");
#endif
}
```

C.2 3D_per.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <complex.h>
4  #include <fftw3.h>
5  #include <time.h>
6  #include "mt.h"
7
8  #define DEBUG    (0)
9
10 #define NORMOUT (1)
11
12 #define N        (64)
13 #define NH       ((N) / 2 + 1)
14 #define NLONG    (128)
15 #define NLONGH   ((NLONG) / 2 + 1)
16
17 #define PI       (3.141592653589793)
18
19 #define sq(x)    ((x)*(x))
20 #define cu(x)    ((x)*sq(x))
21
22 #define coordh(i,j,k)        ((k) + NH * ((j) + N * (i)))
23 #define coordlongh(i,j,k)    ((k) + NLONGH * ((j) + NLONG * (i)))
24 #define coordlong(i,j,k)     ((k) + NLONG * ((j) + NLONG * (i)))
25
```

```
26 void scale(fftw_complex *b, fftw_complex *bext){
27     //Copys data from the array the transform was calculated on
          and scales the
28     //data simultanaeously.
29     register int i,j,k;
30
31     for (i=0;i<N/2;i++){
32         for (j=0;j<N/2;j++) {
33             for (k=0;k<NH-1;k++) {
34                 b[coordh(i,j,k)] = bext[coordlongh(i,j,k)]
35                                    / cu(NLONG);
36                 if (i != 0) {
37                     b[coordh(N-i,j,k)] = bext[coordlongh(NLONG-i,
                          j,k)]
38                                          / cu(NLONG);
39                 }
40                 if (j != 0) {
41                     b[coordh(i,N-j,k)] = bext[coordlongh(i,NLONG-
                          j,k)]
42                                          / cu(NLONG);
43                 }
44                 if (i != 0 && j != 0) {
45                     b[coordh(N-i,N-j,k)] = bext[coordlongh(NLONG-
                          i,NLONG-j,k)]
46                                            / cu(NLONG);
```

```
47                  }
48              }
49          }
50      }
51  }
52
53  void print_complex(fftw_complex *a, int n, int nh){
54      //Prints an n x n x nh array of complex numbers to stdout in
            a way that
55      //util will be able to read
56      register int i,j,k;
57
58      for (i=0;i<n;i++) {
59          printf("[");
60          for (j=0;j<n;j++) {
61              printf("[");
62              for (k=0;k<nh;k++) {
63                  switch (nh) {
64                      case NH:
65                          printf("%lg%+lgj",
66                                  creal(a[coordh(i,j,k)]),
67                                  cimag(a[coordh(i,j,k)]));
68                          break;
69                      case NLONGH:
70                          printf("%lg%+lgj",
```

```
                                    creal(a[coordlongh(i,j,k)]),
                                    cimag(a[coordlongh(i,j,k)]));
                            break;
                    }
                    if (k < nh-1) {
                        printf(",");
                    }
                }
                printf("]");
                if (j < n-1) {
                    printf(",");
                }
            }
            printf("]\n");
        }
}

void print_double(double *a, int n){
    //Prints an n x n x n array of double to stdout in a way that util can
    //decipher.
    register int i,j,k;

    for (i=0;i<n;i++) {
        printf("[");
```

```
95          for (j=0;j<n;j++) {
96              printf("[");
97              for (k=0;k<n;k++) {
98                  printf("%lg", a[coordlongh(i,j,k)]);
99                  if (k < n-1) {
100                     printf(",");
101                 }
102             }
103             printf("]");
104             if (j < n-1) {
105                 printf(",");
106             }
107         }
108         printf("]\n");
109     }
110 }
111
112 double norm(fftw_complex *a){
113     //computes the two-norm in fourier space
114
115     register int i;
116     double value;
117
118     value = 0.0;
119     for (i=0;i<sq(N)*NH;i++){
```

```
120         value += (double)(a[i] * conj(a[i]));
121     }
122     return sqrt(value);
123 }
124
125 void b_update(double *u, double *f, fftw_plan fft, fftw_plan ifft
        ){
126     //computes the non-linearity by computing the ifft, computing
            the
127     //non-linearity and computing the fft of that.
128
129     register int i;
130
131     fftw_execute(ifft);
132
133     for (i=0;i<cu(NLONG);i++){
134         f[i] = u[i] - cu(u[i]);
135     }
136
137     fftw_execute(fft);
138 }
139
140 void a_copy(fftw_complex *a, fftw_complex *aext){
141
142     register int i,j,k;
```

```
143
144     // Zero the array (FFTW will destroy it, so it needs to be
            rezeroed at
145     // everytime step).
146     for (i=0;i<sq(NLONG)*NLONGH;i++){
147         aext[i] = 0;
148     }
149
150     // Copy the smaller array into the padded one so that
            symmetry is
151     // preserved.
152     for (i=0;i<N/2;i++){
153         for (j=0;j<N/2;j++) {
154             for (k=0;k<NH-1;k++) {
155                 aext[coordlongh(i,j,k)] = a[coordh(i,j,k)];
156                 if (i != 0) {
157                     aext[coordlongh(NLONG-i,j,k)] = a[coordh(N-i,
                            j,k)];
158                 }
159                 if (j != 0) {
160                     aext[coordlongh(i,NLONG-j,k)] = a[coordh(i,N-
                            j,k)];
161                 }
162                 if (i != 0 && j != 0) {
```

```
                        aext[coordlongh(NLONG-i,NLONG-j,k)] = a[coordh(N-i,N-j,k)];
                    }
                }
            }
        }
}

main(){

    register int i,j,k,l;
    int initial, numplots, plotstep, normstep;
    double *u, *f, *kappa, *kappap, mu, sigma, lambda, tmp_real, tmp_imag,
            h, timeend;
    fftw_complex *a, *b, *aext, *bext;
    fftw_plan fft, ifft;

    kappa = (double*) fftw_malloc(sizeof(double) * sq(N) * NH);
    kappap= (double*) fftw_malloc(sizeof(double) * N);
    a     = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sq(N) * NH);
    b     = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sq(N) * NH);
```

```
    aext  = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sq
        (NLONG) * NLONGH);
    bext  = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * sq
        (NLONG) * NLONGH);
    u     = (double*) fftw_malloc(sizeof(double) * cu(NLONG));
    f     = (double*) fftw_malloc(sizeof(double) * cu(NLONG));

    fftw_import_system_wisdom();
    fft  = fftw_plan_dft_r2c_3d(NLONG,NLONG,NLONG, f, bext,
        FFTW_EXHAUSTIVE);
    ifft = fftw_plan_dft_c2r_3d(NLONG,NLONG,NLONG, aext, u,
        FFTW_EXHAUSTIVE);

    // Read coefficients from stdin
    scanf("%lf %lf %lf %lf %lf %d",
          &mu, &lambda, &sigma, &h, &timeend, &numplots);
    plotstep = (int) ceil(timeend / h / numplots);
    normstep = (int) ceil(timeend / h / (numplots*10));

    // See if stdin tells us to use an initial condition
    // 0 means no, 1 means yes
    scanf("%d", &initial);

    // Initializtion of eigenvalue array

```

```
204     for (i=0;i<N/2;i++) {
205         kappap[i] =  4.0 * sq(i) * sq(PI);
206         kappap[i+N/2] =  4.0 * sq(N/2-i) * sq(PI);
207     }
208
209     for (i=0;i<N;i++) {
210         for (j=0;j<N;j++) {
211             for (k=0;k<NH;k++) {
212                 kappa[coordh(i,j,k)] = kappap[i] + kappap[j] +
                        kappap[k];
213             }
214         }
215     }
216
217     if (initial) {
218         for (i=0;i<sq(N)*NH;i++) {
219             scanf("%lf %lf", &tmp_real, &tmp_imag);
220             a[i] = tmp_real + I*tmp_imag;
221         }
222     }
223
224     // Use random initial conditions otherwise
225     else {
226         // Initialization of the coefficient array
227         // (first we need a seed)
```

```
        init_genrand(time(NULL));

        // Random coefficients in real space
        for (i=0;i<NLONG;i++) {
            for (j=0;j<NLONG;j++) {
                for (k=0;k<NLONG;k++) {
                    f[coordlong(i,j,k)] = .1 * genrand_realc();
                }
            }
        }

        fftw_execute(fft);

        // Zero a before copy
        for (i=0;i<sq(N)*NH;i++) {
            a[i] = 0.0;
        }

        scale(a,bext);
        a[0] = mu;
    }

    // Initialize b to zeros
    for (i=0;i<sq(N)*NH;i++) {
        b[i] = 0.0;
```

```
253         }
254
255 #if DEBUG
256         printf("a\n");
257         print_complex(a,N,NH);
258
259         a_copy(a,aext);
260
261         printf("\naext\n");
262         print_complex(aext,NLONG,NLONGH);
263
264         fftw_execute(ifft);
265
266         //print_double(u,NLONG);
267
268         for (i=0;i<cu(NLONG);i++){
269             f[i] = u[i];
270         }
271
272         //print_double(f,NLONG);
273
274         fftw_execute(fft);
275
276         printf("\nbext\n");
277         print_complex(bext,NLONG,NLONGH);
```

```
    scale(b,bext);

    printf("\nb\n");print_complex(b,N,NH);

    int works = 1;
    for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
            for (k=0;k<NH;k++) {
                if ((double) ((a[coordh(i,j,k)]-b[coordh(i,j,k)])
                                * conj(a[coordh(i,j,k)]-b[coordh(i,j,
                                    k)]))
                                > 1e-16) {
                    works = 0;
                    printf("\nTransform not invertible, at (%d,%d
                        ,%d),\
                                %lf%+lfi != %lf%+lfi\n",
                                i,j,k,
                                creal(a[coordh(i,j,k)]),cimag(a[coordh
                                    (i,j,k)]),
                                creal(b[coordh(i,j,k)]),cimag(b[coordh
                                    (i,j,k)]));
                }
            }
        }
```

```
299         }
300         if (works) {
301             printf("\nTransform is invertible!\n");
302         }
303
304 #else
305         for (l=0;l*h<timeend;l++) {
306
307             if(l % plotstep == 0) {
308                 printf("begin plot\n");
309                 printf("time %lg\n", l*h);
310                 print_complex(a,N,NH);
311                 printf("end plot\n");
312             }
313 #if NORMOUT
314             if(l % normstep == 0) {
315                 printf("norm %lg\n",norm(a));
316             }
317 #endif
318             a_copy(a,aext);
319             b_update(u,f,fft,ifft);
320             scale(b,bext);
321
322             for (i=0;i<N;i++) {
323                 for (j=0;j<N;j++) {
```

```
                    for (k=0;k<NH-1;k++) {
                        if ((N%2 != 0 || (i != N/2 && j != N/2)) &&
                            !(i == 0 && j == 0 && k == 0)) {
                            a[coordh(i,j,k)] = (a[coordh(i,j,k)]
                                + h * lambda * b[coordh(i,j,k)] *
                                    kappa[coordh(i,j,k)])
                                / (1 + h * sq(kappa[coordh(i,j,k)]) +
                                    h * sigma * lambda);
                        }
                    }
                }
            }
        }
        printf("begin plot\n");
        printf("time %lf\n", timeend);
        print_complex(a,N,NH);
        printf("end plot\n");
#endif
}
```

Bibliography

Bibliography

[1] Frank S. Bates and Glenn H. Fredrickson. Block copolymer thermodynamics: Theory and experiment. *Annu. Rev. Phys. Chem.*, 41:525–57, 1990.

[2] Rustum Choksi, Mirjana Maras, and J. F. Williams. 2d phase diagram for minimzers of a nonlocal cahn-hilliard functional. *SIAM J. Appl. Dynamical Systems*, 2010.

[3] Rustum Choksi, Mark A. Peletier, and J. F. Williams. On the phase diagram for microphase separation of diblock copolymers: An approach via a nonlocal Cahn-Hilliard functional. *SIAM J. Appl. Math.*, 69(6):1712–1738, 2009.

[4] Lawrence C. Evans. *Partial Differential Equations.* American Mathematical Society, 1998.

[5] Paul C. Fife. Models for phase separation and their mathematics. *Electronic Journal of Differential Equations*, 2000(48):1–26, 2000.

[6] Matteo Frigo and Steven G. Johnson. *The Design and Implementation of FFTW3*, 2005.

[7] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations.* Cambridge University Press, 2 edition, 2009.

[8] Michael Droettboom John Hunter, Darren Dale. *matplotlib User's Guide.*

[9] A.N. Kolmogorov and S.V. Fomin. *Introductory Real Analysis.* Dover, 1975.

[10] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[11] Takao Ohta and Kyozi Kawasaki. Equilibrium morphology of block copolymer melts. *Macromolecules*, 19(10):2621–2632, 1986.

[12] Pearu Peterson. *PyVTK – Tools for manipulating VTK files in Python.*

[13] Python Software Foundation. *Python v2.7.1 documentation.*

[14] Takashi Teramoto and Yasumasa Nishiura. Double gyroid morphology in a gradient system with nonlocal effects. *Journal of the Physical Society of Japan*, 71(7):1611–1614, 2002.

[15] Takashi Teramoto and Yasumasa Nishiura. Double gyroid morphology of the diblock copolymer problem. *RIMS Kokyuroku*, 1356:116–121, 2004.

Curriculum Vitae

Michael Atkins graduated from McLean High School, McLean, Virginia in 2006. He received his Bachelor of Science in Mathematics from George Mason University in 2010. During this time, he was employed by the Department of State as a Computer technician and the National Institute of Standards and Technology as a researcher.