Computational Exploration of Vortex Nucleation In Type II Superconductors

Using a Finite Element Method in Ginzburg-Landau Theory

Alden Pack

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Mark Transtrum, Advisor
Eric Hirschmann
Michael Scott

Department of Physics and Astronomy

Brigham Young University

December 2017

ABSTRACT

Computational Exploration of Vortex Nucleation In Type II Superconductors
Using a Finite Element Method in Ginzburg-Landau Theory

Alden Pack
Department of Physics and Astronomy, BYU
Master of Science

Using a finite element method, we numerically solve the time-dependent Ginzburg-Landau equations of superconductivity to explore vortex nucleation in type II superconductors. We consider a cylindrical geometry and simulate the transition from a superconducting state to a mixed state. Using saddle-node bifurcation theory we evaluate the superheating field for a cylinder. We explore how surface roughness and thermal fluctuations influence vortex nucleation. This allows us to simulate material inhomogeneities that may lead to instabilities in superconducting resonant frequency cavities used in particle accelerators.

Keywords: Ginzburg-Landau, superconductor, finite element method

ACKNOWLEDGMENTS

To my wife who enlightens my days and helps me be a better person.

I'd like to thank Mark Transtrum for his patience and guidance as we worked on this challenging problem together.

This work was supported by the U.S. National Science Foundation under Award PHY-1549132, the Center for Bright Beams.

Contents

Chapter 1

Introduction

1.1 Significance of Research

Imaging is an essential component of medicine, engineering, and science. The smaller the object the more difficult the acquisition of a clear image. Biologists studying proteins, engineers developing semiconductor technology, and physicists studying magnetic materials all need bright, coherent, and tunable x-ray beams for their research [1–3]. A common source for x-ray beams is synchrotrons. Unfortunately the current size and costs of these accelerators restricts their accessibility to researchers.

This work is part of a collaboration with the Center for Bright Beams (CBB), an NSF funded science and technology center, which seeks to increase the quality and decrease the cost of beams produced by accelerators. There are three main areas of improvement targeted by this center: beam creation, beam acceleration, and beam storage. Our efforts are directed at the field of beam acceleration. Specifically we wish to explore one of the limiting factors to the performance of superconducting resonance cavities: the quench due to large induced magnetic fields.

Beam acceleration is the process of speeding up charged particles to relativistic velocities.

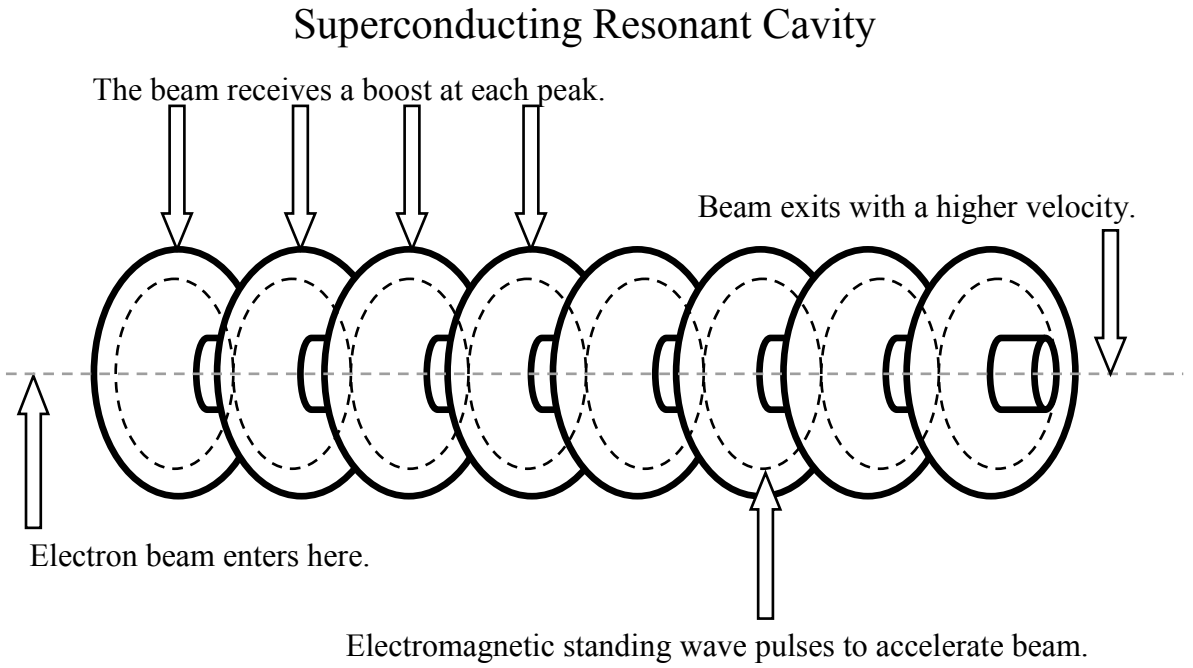


Figure 1.1 A series of superconducting radio frequency cavities are connected together. The inside of the cavities are plated with niobium, a superconductor. An AC current is tuned so that an entering bunch of electrons is accelerated at the center of each cavity.

This is done in accelerators by using superconducting radio frequency (SRF) cavities [4]. Fig. 1.1 portrays what SRF cavities look like. This image shows a series of cavities all connected together. The interiors of the cavities are plated with a superconductor, such as niobium. An AC current running through the cavities creates internal electromagnetic fields. The frequency of the AC current is tuned such that an entering cluster of electrons receives a boost of energy at the center of each cavity. The electrons are then used as an electron beam, or deflected to produce x-rays.

Much of the costs of operating SRF cavities comes from large cryogenic facilities that cool them to around 2 degrees kelvin, well below the boiling point of liquid helium. The largest electromagnetic fields and highest frequencies are achieved at this temperature, producing the brightest beams. Raising this operating temperature by improving SRF cavity stability would eliminate large

portions of the cryogenic facilities, decreasing the size and cost of maintaining accelerators.

Superconducting materials like Nb and Nb_3Sn face limitations from material inhomogeneities [5, 6]. Upon transitioning to a superconducting state magnetic fields can be trapped by imperfections. Applied AC currents move the trapped fields, thereby dissipating heat and lowering cavity quality. Some of these inhomogeneities include grain boundaries, surface roughness, and variations in Sn concentrations. Experts in cavity design need to know which material inhomogeneities are the most influential in reducing cavity stability.

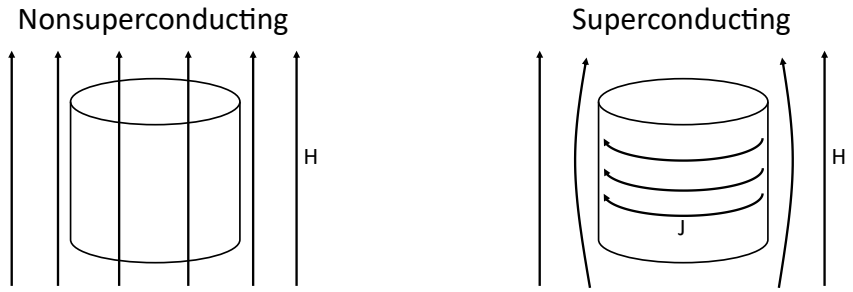
Building and testing SRF cavities is expensive and the physics behind dynamic superconductivity is difficult to measure. By using numerical methods we can paint a picture of what should happen experimentally. We can simulate how material inhomogeneities influence accelerator performance and guide development efforts.

1.2 Theory and Previous work

1.2.1 A Brief Review of Superconductivity

Superconductors have two hallmark phenomena: negligible DC resistance and the Meissner effect (expulsion of magnetic fields) [7]. The negligible DC resistance in superconductors was first found by Kamerlingh Onnes in 1911 [8]. This property is ideal for applications requiring large currents. Note that AC currents have a small but nonnegligible resistance. The Meissner effect is due to the formation of surface currents which induce magnetic fields that cancel out the external field (as illustrated in Fig. 1.2). A material loses superconductivity if it exceeds the critical temperature T_c or if the magnetic fields are so strong they break through the Meissner effect. For SRF cavities the small AC resistance and negligible DC resistance are desired traits while the Meissner effect is a limitation.

The mechanistic origin of superconductivity is the creation of Cooper pairs: 2 electrons of



(a) **(b)**

Figure 1.2 The expulsion of magnetic fields from within a superconductor is known as the Meissner effect. In a nonsuperconducting state magnetic field lines are free to go through the material. In a superconducting state surface currents form that induce magnetic fields opposite to the external magnetic field, leaving zero magnetic field in the superconductor.

opposite spin and momentum experience a positive attraction through the interaction of phonons (lattice vibrations). Negligible resistance and the Meissner effect arise as the Cooper pairs, which are bosons, form a Bose-Einstein condensate.

1.2.2 Ginzburg-Landau Theory: An Appropriate Model

There are many models of superconductivity [7]. Eliashberg theory and BCS theory consider interactions of individual Cooper pairs. The simpler London model captures macroscopic features of superconductivity but is valid only for small variations in the magnetic field. We choose to use the Ginzburg-Landau equations as they capture macroscopic features including vortex dynamics and the difference between type I and type II superconductors.

Phenomenological in origin, Ginzburg and Landau proposed an ansatz for the form of the free energy difference of the superconducting and nonsuperconducting state. By minimizing this free energy difference one could derive partial differential equations that give the macroscopic

behavior of superconductors. They hypothesized the free energy could be approximated by a Taylor expansion of a complex order parameter (whose norm squared is the local superconducting electron density) centered around T_c.

Du et al. review how to derive the Ginzburg-Landau equations (GL) of superconductivity assuming no spatial variations of the critical magnetic field H_c and T_c [9]. We will follow their notation. The difference between the free energy in a normal state and a superconducting state is given by

$$G(\psi, \mathbf{A}) = \int_\Omega \Big(f_n - |\psi|^2 + \frac{1}{2}|\psi|^4 + \Big| \Big(-\frac{i}{\kappa}\nabla - \mathbf{A} \Big) \psi \Big|^2 + |\mathbf{h}|^2 - 2\mathbf{h}\cdot\mathbf{H} \Big),$$

where G, the Gibbs free energy, depends on the complex order parameter ψ and the magnetic vector potential $\mathbf{A}$. Units are chosen such that if the norm squared of ψ is one then the state is superconducting, otherwise it loses superconductivity as ψ approaches zero. The magnetic fields has units of $\sqrt{2}H_c$. f_n is the free energy of a superconductor in the normal state and κ, a dimensionless constant, is the ratio of the penetration depth λ and the coherence length ξ. The penetration depth is the distance at which the magnetic field falls off inside a superconductor. Similarly, the coherence length is the scale over which the order parameter can vary substantially. The total magnetic field $\mathbf{h}$ is given by $\nabla \times \mathbf{A}$ and $\mathbf{H}$ is the applied magnetic field. Ω is the superconducting region and we will denote the boundary of that region as Γ. Note that distance is measured in terms of penetration depth.

Solving for where the first variation of G is zero one can derive the Ginzburg Landau equations of superconductivity (GL) given by

$$\Big(-\frac{i}{\kappa}\nabla-\mathbf{A}\Big)^2\psi-\psi+|\psi|^2\psi=0 \text{ in } \Omega$$
$$\nabla\times(\nabla\times\mathbf{A})=-\frac{i}{2\kappa}(\psi^*\nabla\psi-\psi\nabla\psi^*)-|\psi|^2\mathbf{A}+\nabla\times\mathbf{H} \text{ in } \Omega$$
$$\Big(\frac{i}{\kappa}\nabla\psi+\mathbf{A}\psi\Big)\cdot\mathbf{n}=0 \text{ on } \Gamma$$
$$(\nabla\times\mathbf{A})\times\mathbf{n}=\mathbf{H}\times\mathbf{n} \text{ on } \Gamma.$$

Solving the GL equations gives the state that minimizes G.

Eliashberg later generalized GL to include time dependence [10]. The time-dependent Ginzburg-Landau equations (TDGL) are written below. With time dependence we must consider the electric potential ϕ, the electric field $\mathbf{E}=-\nabla\phi-\frac{\partial\mathbf{A}}{\partial t}$, and a time constant η. We set η to one for convenience.

$$\eta\frac{\partial\psi}{\partial t}+i\eta\kappa\phi\psi+\Big(\frac{i}{\kappa}\nabla+\mathbf{A}\Big)^2\psi-\psi+|\psi|^2\psi=0 \text{ in } \Omega$$
$$\frac{\partial\mathbf{A}}{\partial t}+\nabla\times(\nabla\times\mathbf{A})+\nabla\phi+\frac{i}{2\kappa}(\psi^*\nabla\psi-\psi\nabla\psi^*)+|\psi|^2\mathbf{A}=\nabla\times\mathbf{H} \text{ in } \Omega$$
$$\Big(\frac{i}{\kappa}\nabla\psi+\mathbf{A}\psi\Big)\cdot\mathbf{n}=\mathbf{H}\times\mathbf{n} \text{ on } \Gamma$$
$$(\nabla\times\mathbf{A})\times\mathbf{n}=\mathbf{H}\times\mathbf{n} \text{ on } \Gamma$$
$$\mathbf{E}\cdot\mathbf{n}=0 \text{ on } \Gamma$$

We will focus on the time-dependent case and use the time-independent problem to verify the accuracy of steady state solutions.

1.2.3 Phases in Superconductors.

The utility of Ginzburg-Landau theory is the ability to macroscopically describe superconductivity. We can treat G as a high dimensional surface where the solutions of the GL equations are extrema

of G. By inspection one can see the TDGL equations are diffusive, meaning in a nonequilibrium state the system will evolve to a state of minimal energy (a minimum of G). The global minimum of G is the stable state while local minima are metastable states. As $\mathbf{H}$ changes the minima of G may change. A local minimum can turn into the global minimum and vice versa. As an example, for small $\mathbf{H}$ the global minimum of G is the superconducting state (assuming the system is below T_c) whereas for high $\mathbf{H}$ the global minimum is a nonsuperconducting state. A critical magnetic field is a field that causes a change of the global minimum. Within TDGL theory metastable states can transition to a stable state if thermal fluctuations overcome the energy barrier [11].

The intermediate behavior between superconducting and nonsuperconducting states is determined by κ. Type I superconductors are characterized by $\kappa < \frac{1}{\sqrt{2}}$ while type II superconductors have $\kappa > \frac{1}{\sqrt{2}}$. Transtrum numerically created Fig. 1.3 based on an infinite superconducting slab (half of space filled with vacuum and half of space filled with a superconducting material) [12]. As the external magnetic field increases a type I superconductor has only one critical field strength H_c. Type II superconductors have two critical field strengths, H_{c1} and H_{c2}. Between these two field strengths can be a mixed state. In this state it is energetically favorable for vortices, filaments of magnetic field, to enter the surface of the material and form a vortex lattice. These vortices are nonsuperconducting in the center. As they move through the material, they will dissipate heat (see Fig. 1.4). The second critical field marks where all superconductivity is lost.

We are interested in the transition from superconducting states to mixed states for type II superconductors. The largest field an SRF cavity can operate in without vortices are metastable states where the system is superconducting, but at magnetic fields greater than the critical field. The superheating field H_{sh} marks where the metastable superconducting states no longer exist. This corresponds to the vanishing of the energy barrier that prevented the system from transitioning to the mixed state [13]. We will simulate the physics near H_{sh} as this is the largest field attainable by a Type II superconductor.

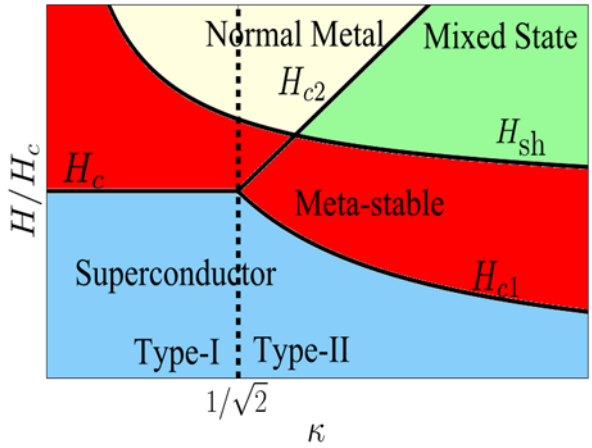


Figure 1.3 Within Ginzburg-Landau theory type I superconductors are characterized by $\kappa < \frac{1}{\sqrt{2}}$ while type II superconductors have $\kappa > \frac{1}{\sqrt{2}}$. Type I superconductors have a single critical magnetic field H_c while type II have two: H_{c1} and H_{c2}. Between those two critical magnetic fields is a mixed state of both superconducting and nonsuperconducting regions. Vortices, or filaments of magnetic field, reside in the nonsuperconducting regions. It is possible to be in a metastable state with magnetic fields greater than the critical field (known as the superheating field) as an energy barrier must be crossed for state transitions to occur.

Note that because of the high dimensionality of G the transition from superconducting to mixed state involves symmetry breaking. This means there are more than one path that an unstable state can take within TDGL theory for the system to minimize the free energy. For more details see [13].

1.2.4 Saddle-Node Bifurcation Theory and Free Energies

As defined in the previous section, H_{sh} is the magnetic field at which the energy barrier between a metastable superconducting state and the mixed state vanishes. The top of the barrier is unstable while the bottom of the metastable state is locally stable. Upon raising the field beyond H_{sh} the unstable point and stable point annihilate each other so that the system has one minimum, the mixed state. This annihilation of a stable and unstable point is the defining characteristic of a

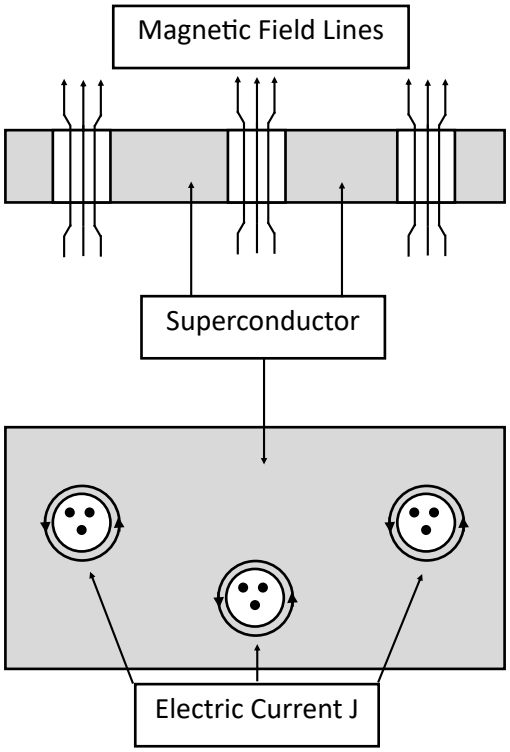


Figure 1.4 In the mixed state of type II superconductors filaments of magnetic field known as vortices form. Their centers are nonsuperconducting and their movement dissipates heat. Each vortex is surrounded by electric current.

saddle-node bifurcation [14]. We review this theory.

Saddle-node bifurcation theory says that near the annihilation of the stable and unstable point the system locally behaves as

$$\frac{dx}{dt} = r - x^2,$$

where x is the state variable (some combination of ψ and $\mathbf{A}$). The value r is the bifurcation parameter. The energy barrier exists for $r > 0$, does not exist for $r < 0$, and disappears at $r = 0$. Clearly $r = 0$ when $|\mathbf{H}| = H_{sh}$. We can then expect that r depends on some combination of $H_{sh} - |\mathbf{H}|$. This formulation will become invaluable when trying to understand what happens when $\mathbf{H}$ is very close to H_{sh}.

1.2.5 The Finite Element Method

To solve the TDGL equations and the GL equations we choose to use the finite element method (FEM) as there are no closed-form solutions on complicated geometries. This method discretizes the domain of interest into finite elements that are fitted to approximate solutions to partial differential equations. Hughes' book is a good introduction to this method [15].

Several authors have analyzed the convergence of FEM with the GL equations and the TDGL equations [9, 16, 17]. A particularly ingenious formulation allows finite element solutions of the TDGL equations to converge on any curved polyhedral domain [18, 19]. The authors reformulate the problem into a series of Poisson and Laplace equations. We will use this approach. For the time-independent problem we will use Du's approach [17].

1.3 Research Goals

We mentioned that the superheating field occurs from metastability and the need to cross an energy barrier. Vortices form once that barrier is crossed. Transtrum et al. used linear stability analysis to find the perturbative wavelength which leads to this transition [13]. It was found that this wavelength differs from the final spacing of vortices within the medium. The dynamics from initial nucleation to final vortex spacing are unknown. By running time-dependent simulations we demonstrate aspects of that evolution.

Several of our collaborators at CBB are interested in knowing the optimal smoothness for SRF cavities. We will partially address this issue by simulating how surface roughness influences vortex dynamics.

Finally, we apply saddle-node bifurcation theory to our simulations in order to estimate H_{sh} within the TDGL equations. This provides us with a way to calculate the largest possible applied field without the nucleation of vortices.

1.4 Organization of this Thesis

In chapter 2 we introduce our methodology. We introduce our FEM solver and the geometry of our problem. We describe mesh generation for both a symmetric circle and a rough circle. Finally, we describe how to use saddle-node bifurcation theory to estimate H_{sh}.

In chapter 3 we present our results. We create simulations above and below H_{sh} and we show how adding surface roughness causes vortex nucleation at lower fields. We simulate thermal fluctuations by introducing noise into the system. We then estimate H_{sh} for the symmetric cylinder and show how the choice of timestep and finite element type influence that estimate. We also show how H_{sh} varies with κ and compare our results with previous work done by Transtrum [12]. We also give H_{sh} evaluated for a rough cylinder. Finally, we compare the solution from GL to the steady

state solution for the TDGL equations.

Chapter 4 wraps up the thesis by giving a brief overview of the importance of what we have accomplished. We then propose several avenues this research can take to further explore material inhomogeneities and other complicated geometries.

Chapter 2

Methodology

2.1 FEniCS and the Finite Element Method

As mentioned in the introduction, we use the Finite Element Method to solve the TDGL equations and the GL equations. We use FEniCS as our finite element solver. More information on this program can be found in [20]. For changes in implementation visit the FEniCS website, https://fenicsproject.org.

We follow Li and Zhangs' approach to discretizing the TDGL equations using FEM [18]. This approach reduces the TDGL system to a series of Laplace and Diffusion equations. This allows us to use piecewise linear (Lagrange) finite elements on general curved polyhedra in two dimensions. We discretize time through a decoupled backward Euler method. The domain is described in the next section. For the time-independent problem we will use Du's approach [17].

2.2 Geometry

A simple physical system to consider is a cylindrical superconducting wire. We consider an infinitely long cylinder without variations along the z-axis (see Fig. 2.1). We also assume the applied

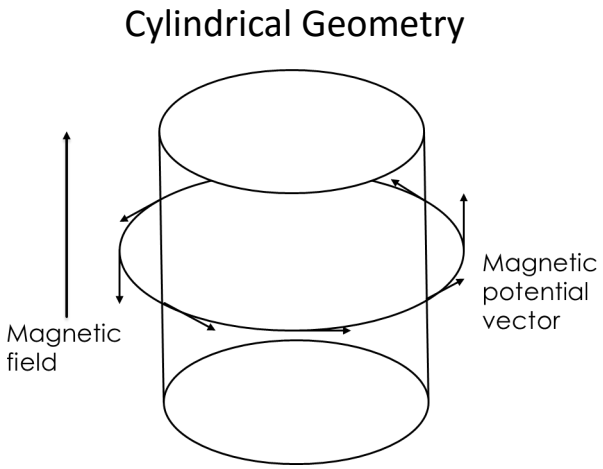


Figure 2.1 In the cylindrical geometry we have symmetry along the z-axis such that variations occur in the x-y plane. The magnetic field is applied parallel to the z-axis.

magnetic field is parallel to the z-axis, $\mathbf{H} = Ha\hat{z}$ where Ha is a value that may depend on time. We can then run simulations on a circular cross-section as the solution is independent of z. This domain provides us with enough degrees of freedom to simulate vortex nucleation without becoming too computationally cumbersome.

One of our goals is to see how surface roughness influences vortex nucleation. We will compare a smooth surface with a rough one. For the smooth cylinder we've found that symmetry in mesh

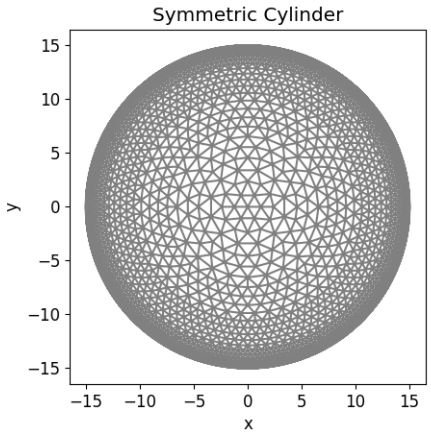


Figure 2.2 This is the mesh for a symmetric cylinder. It has a radius of 15 penetration depths and vertices are uniformly distributed on concentric rings. There are more rings near the boundary as that is where vortex nucleation occurs.

vertex locations improves simulation stability. Near the superheating field, variations in vertex density and rough edges lead to vortex nucleation. To enforce symmetry we divide the circular domain into smaller concentric circles of varying radii. We add vertices uniformly on each circle and then triangulate the domain. For our domain we are mostly interested in dynamics close to the boundary, so we include more circles with large radii to increase the number of vertices near the boundary. Through trial and error we've found that the optimal mesh for running quick yet accurate simulations is one with a radius of 15 penetration depths and 25 concentric circles (see Fig. 2.2).

To add roughness we create noise on the surface of the circle. This noise is a linear combination of sines and cosines with random weights. Since the surface roughness breaks symmetry we let FEniCS automatically add vertices uniformly throughout the domain (see Fig. 2.3).

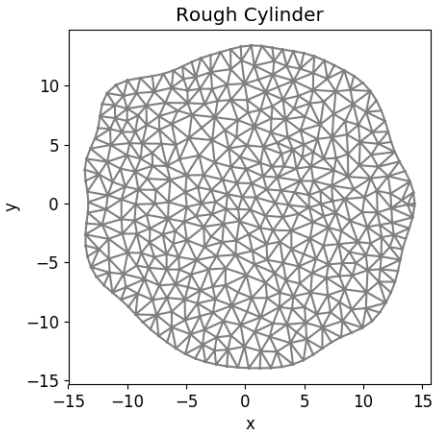


Figure 2.3 This is the mesh for a rough cylinder. Note that the roughness breaks symmetry so we can let FEniCS automatically add vertices uniformly in the domain.

2.3 Extrapolating H_{sh} with Bifurcation Theory

Another of our goals is to find H_{sh}, the superheating field. We previously discussed how the normal form in saddle-node bifurcation theory captures the dynamics of the superconducting phase transition near H_{sh}. The normal form is written as

$$\frac{dx}{dt} = r - x^2$$

where x is the state variable (some combination of ψ and $\mathbf{A}$) and r is the bifurcation parameter. Near a steady metastable state $\frac{dx}{dt} \approx 0$ or $x^2 \approx r$ and $r > 0$. The quantity $\tilde{x} = x - \sqrt{r}$ is a measure of how far off we are from the steady state. If we add a small amount of thermal noise to the steady state then $\tilde{x}$ corresponds to that noise. We can rewrite this as $x = \tilde{x} + \sqrt{r}$. We then get

$$\begin{aligned}\frac{dx}{dt} &= \frac{d\tilde{x}}{dt} \\ &= r - (\tilde{x} + \sqrt{r})^2 \\ &= -\tilde{x}^2 - 2\tilde{x}\sqrt{r}.\end{aligned}$$

As $\tilde{x}$ is small we can drop the first term and get

$$\frac{d\tilde{x}}{dt} = -2\sqrt{r}\tilde{x},$$

which can be solved to give $\tilde{x} = C_0 e^{-2\sqrt{r}t}$ where C_0 is some constant. We see that $\tilde{x}$, added thermal noise to the steady state, decays exponentially with a decay rate $\tau = \frac{1}{2\sqrt{r}}$.

We can see that when Ha is barely less than H_{sh} ($r \approx 0$) that $\tilde{x}$ will decay very slowly. As a matter of fact, as $Ha \to H_{sh}$, $r \to 0$, and $\tau \to \infty$. To get around this we evaluate τ for various Ha, convert that to r, and then extrapolate to when $r = 0$. This gives us when $H_{sh} = Ha$, thus an estimate for H_{sh}.

To find τ we consider the value of $\tilde{x}$ at two time points, t_1 and t_2. Taking the ratio of $\tilde{x}(t_2)$ and $\tilde{x}(t_1)$ yields $\frac{\tilde{x}(t_2)}{\tilde{x}(t_1)} = e^{\frac{t_1 - t_2}{\tau}}$. Solving for τ we find $\tau = \frac{t_1 - t_2}{log(\frac{\tilde{x}(t_2)}{\tilde{x}(t_1)})}$.

As will be seen in chapter 3, added noise does not decay uniformly. Similar to how vibrations on a drum can be decomposed into vibrational modes with varying decay rates, so can the added thermal noise of a superconductor be decomposed into modes. The mode that decays the slowest is the mode that corresponds to vortex nucleation. In other words the slowest decaying mode corresponds to the direction of vanishing energy barrier. This mode is $\tilde{x}$. We use the decay rate of $\tilde{x}$ to find H_{sh}.

Chapter 3

Results

Our three goals mentioned in chapter 1 are to simulate vortex nucleation, estimate H_{sh}, and simulate roughness. We mention our results following that order.

3.1 Simulations Above and Below H_{sh}

To see the difference between states below and above H_{sh} we run time dependent simulations that start in the superconducting state and raise the magnetic field to some fixed value. As we wish to see vortex nucleation the TDGL equations are the appropriate equations to solve.

Let's start with what happens to simulations on the symmetric cylinder above and below H_{sh}. We start in a superconducting state and raise the applied field as $Ha(t) = Hmax(1.0 - e^{-t})$ where $Hmax$ is the maximum applied field. This choice for time dependence allows us to initially raise the field quickly when the dynamics are unimportant, and then slow down as we approach the field strength of interest, $Hmax$.

Remember that the superheating field depends on κ, the ratio of the penetration depth and the coherence length (see Fig. 1.3). To analyze a type II superconductor we choose $\kappa = 4$. Consider what happens for $Hmax = 0.7$ and compare it to the results for $Hmax = 0.785$.

The dynamics for $Hmax = 0.7$ can be seen in Fig. 3.1. At various time steps we plot the norm squared of the order parameter $|\psi|^2$, the magnetic field $\mathbf{h}$, the supercurrent $\mathbf{j}$, and $|\psi|^2$ on the surface of the cylinder. We can see that as time increases $|\psi|^2$ decreases near the boundary. At the same time we see that the magnetic field increases inside the superconductor near the boundary. This indicates that the density of superconducting electrons decreases as the magnetic field increases. The supercurrent increases near the boundary to keep out the external magnetic field. Also note that $|\psi|^2$ on the surface is uniform. $Att = 75$ the system ceases to change and we have reached a stable steady state.

The dynamics for $Hmax = 0.785$ differ significantly as seen in Fig. 3.2. $|\psi|^2$ on the surface decreases until it loses uniformity, becomes zero periodically on the boundary, and then increases. Each low region of $|\psi|^2$ is also a location for large $\mathbf{h}$. These are the vortices. Note that each vortex has the same size and field strength. We have simulated how a superconductor transitions from a superconducting to a mixed state. Clearly $Hmax = 0.74 < H_{sh}$ while $Hmax = 0.785 > H_{sh}$.

3.2 Thermal Fluctuations, Bifurcation, and Finding H_{sh}

We expect that below the superheating field, which is a stable state, that added thermal fluctuations will decay. If we are close to the superheating field or in an unstable state then noise should cause vortices to form. One could procedurally look for H_{sh} by raising the field, waiting for the system to enter a steady state, and then add noise to see if the noise decays. Eventually one would add an infinitesimal amount of noise infinitesimally close to H_{sh} until they reach the precision they desire. Unfortunately the closer one gets to H_{sh}, the longer the system takes to equilibrate. Using saddle-node bifurcation theory we can avoid this conundrum.

In chapter 2 we derived an expression for $\tilde{x}$, a small quantity that decays at different rates depending on a bifurcation parameter. To see how our thermal fluctuations match the behavior of $\tilde{x}$

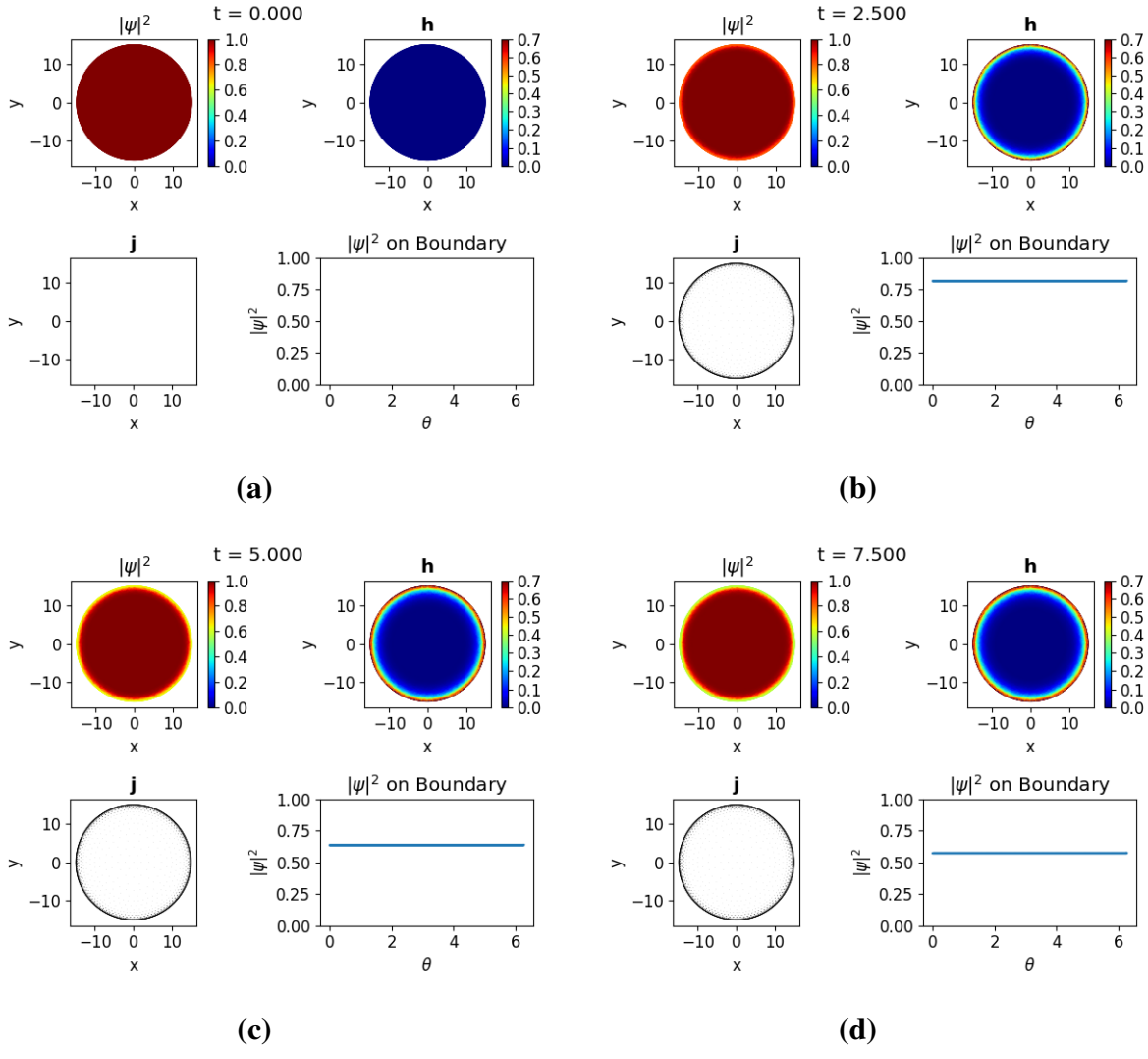


Figure 3.1 As we raise the field to $Hmax = 0.7$ we observe the order parameter decreases near the boundary and the magnetic field increases. At $t = 75$ the system reaches a steady state. This indicates we are below the superheating field.

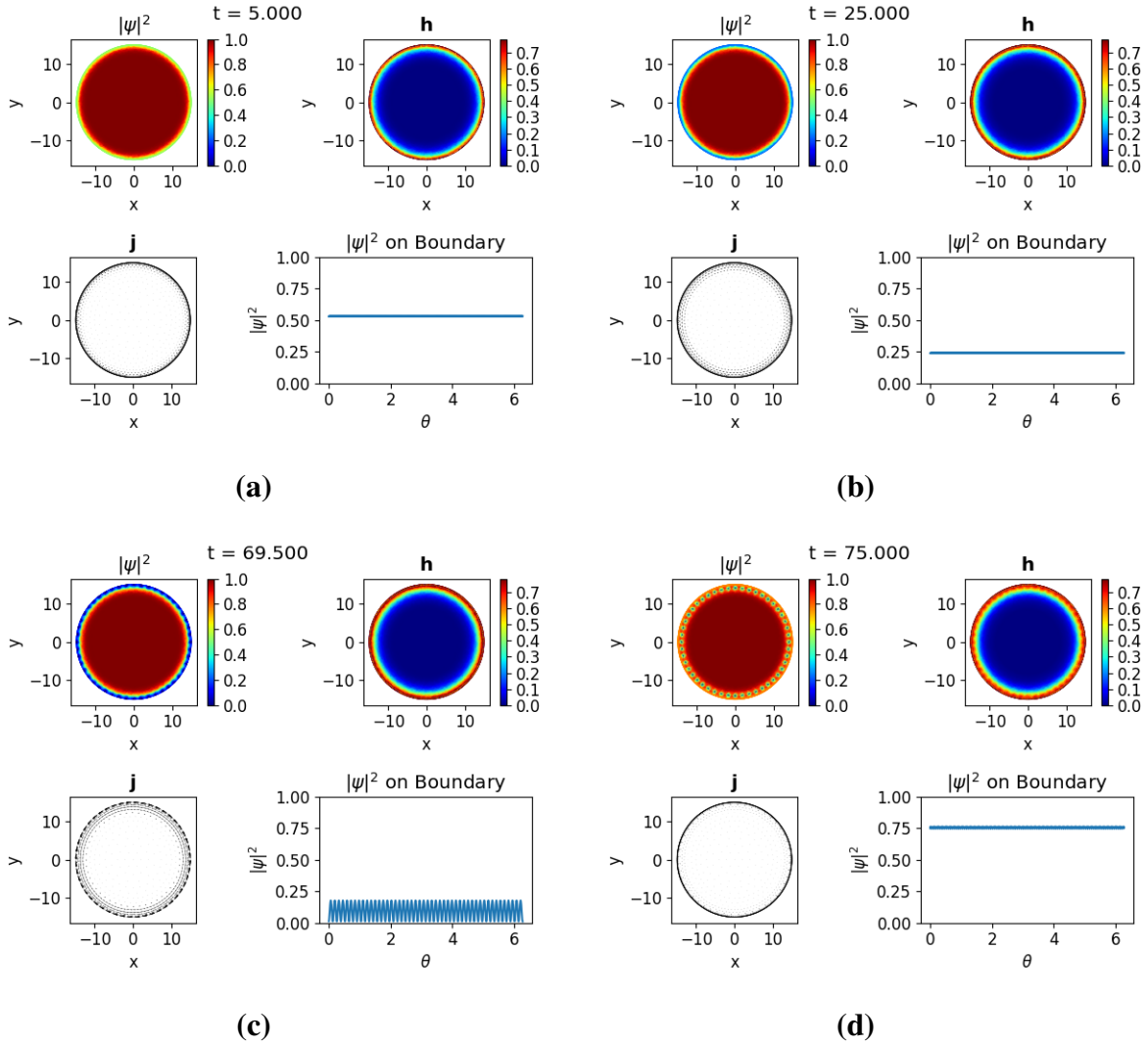


Figure 3.2 As we raise the field to $Hmax = 0.785$ the order parameter decreases until it loses uniformity, becomes zero periodically, and then increases. Vortices begin penetrating the surface at $t = 69.5$ and are fully resolved at $t = 75$. The penetration of vortices means the Meissner state is unstable.

consider Fig. 3.3, where we have run the simulation to the steady state for $Hmax = 0.7$ and added noise, $d|\psi|^2$. We can see that the random noise added to $|\psi|^2$ dies off very quickly in the middle of the cylinder. Near the edge of the cylinder a periodic pattern of high and low values for $d|\psi|^2$ develops. The noise near the boundary dies off slower than the rest of the system. This pattern is the combination of parameters that causes the system to transition from a superconducting state to a mixed state. We have found $\tilde{x}$.

By picking a vertex where this mode of decay is high we calculate how noise on this vertex decays and calculate the decay rates for varying applied fields. Fig. 3.4 shows the location of the vertex we have chosen to observe, indicated by a red dot. We purposely chose a region where this decaying mode is large. Fig. 3.5 shows how noise on this vertex decays as we vary the applied field. There is initially some transient dynamics due to regions of high and low noise interacting, but once the majority of the noise has died, the remaining noise, $\tilde{x}$, decays exponentially. The rate of decay for the final exponential behavior increases as we raise $Hmax$. We can use the tools of section 2.3 to calculate τ and r. In Fig. 3.6 we use a quadratic fit to extrapolate the bifurcation parameter to where it becomes zero, which is where $Hmax = H_{sh}$. We estimate $H_{sh} = 0.738333$.

Transtrum et al. evaluated H_{sh} for varying κ and found that $H_{sh} = 0.7224$ for $\kappa = 4.037$ [13]. In their approach they considered a bulk superconductor (space half filled with vaccum and half filled with superconductor) while we considered a cylinder. We attribute the difference in H_{sh} to this difference in geometry.

A table of values comparing the results of this work with Transtrum's work is given in table 3.1. Note that as κ increases H_{sh} decreases. The relative difference is also calculated. In the table the vertices chosen for evaluating H_{sh} were based on the lowest value of $\tilde{x}$. The previous value of H_{sh} was evaluated on a vertex picked out visually. Hence the difference in value.

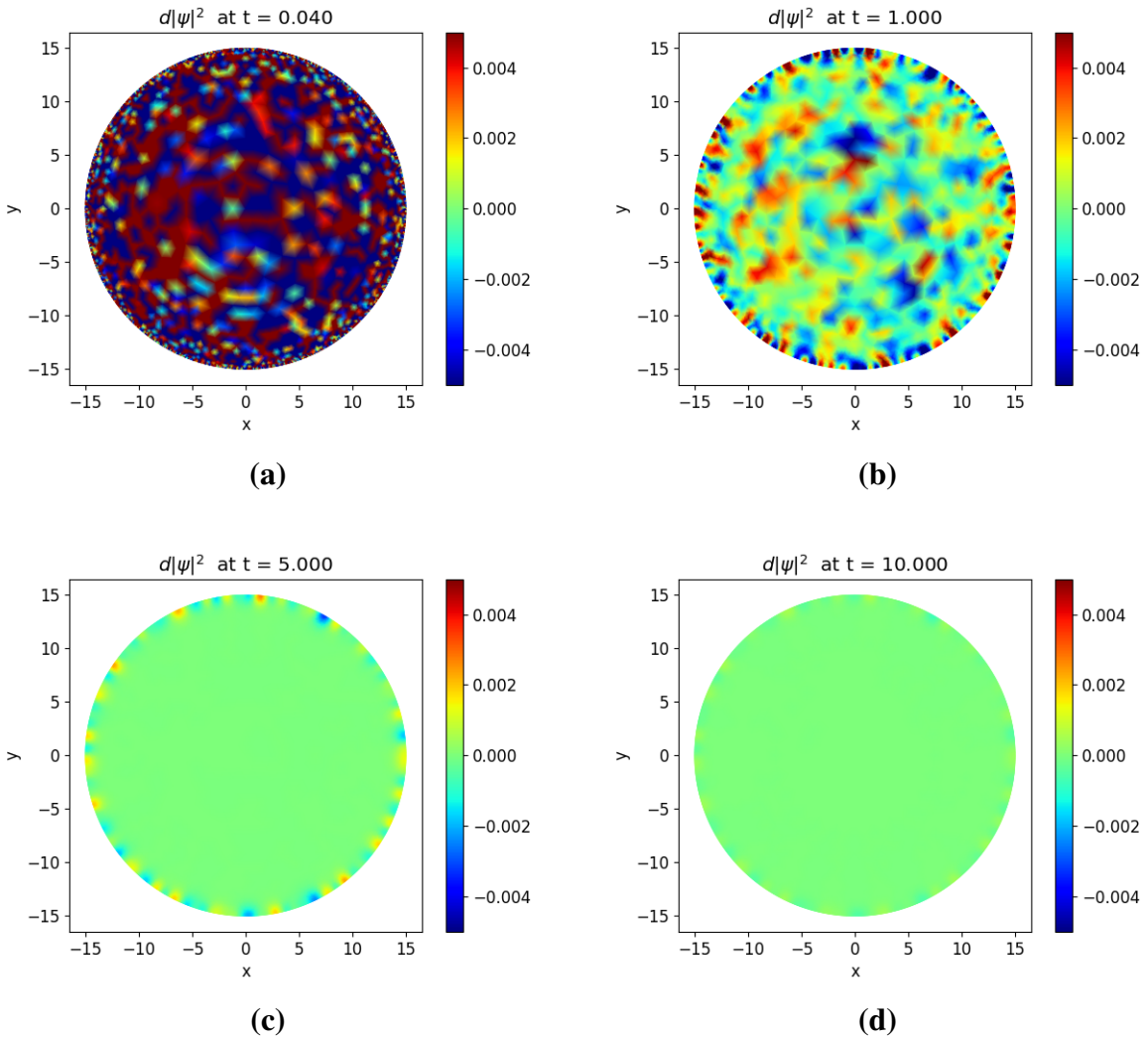


Figure 3.3 We add noise to the superconducting cylinder in the steady state with an applied field of 0.7. Noise in the center of the cylinder decays quickly. Noise at the edges persist in a periodic pattern of low and high regions. This mode is what initiates vortex nucleation. In these images this mode disappears, indicating the system is stable.

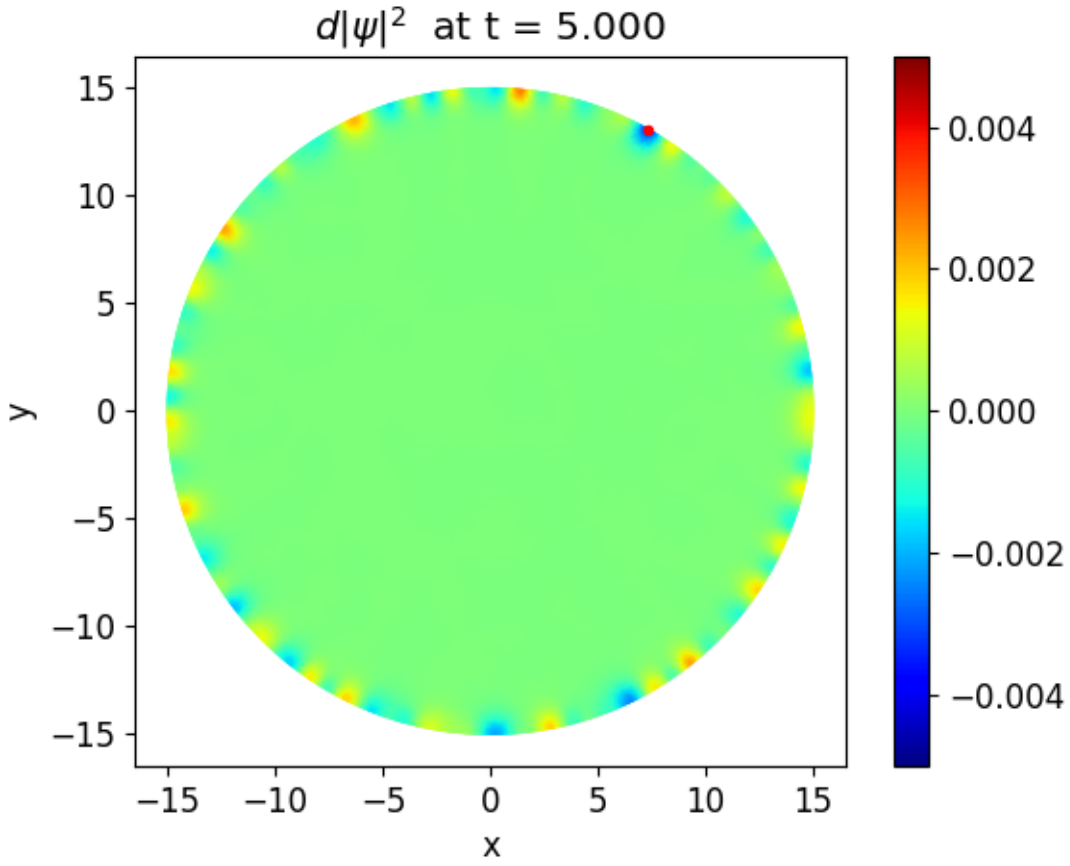


Figure 3.4 The red dot indicates a vertex in a region of large negative noise. Calculating the decay rate of noise at this vertex gives us a way to measure the decay rate of this mode.

	Cylinder H_{sh}	Slab H_{sh}	Relative Difference
$\kappa = 4$	0.7427	0.7224	0.0281
$\kappa = 5$	0.7447	0.7015	0.0616
$\kappa = 6$	0.7181	0.6880	0.0438
$\kappa = 8$	0.7056	0.6653	0.0606

Table 3.1 Variations in H_{sh} depending on κ. Note that as κ increases H_{sh} decreases.

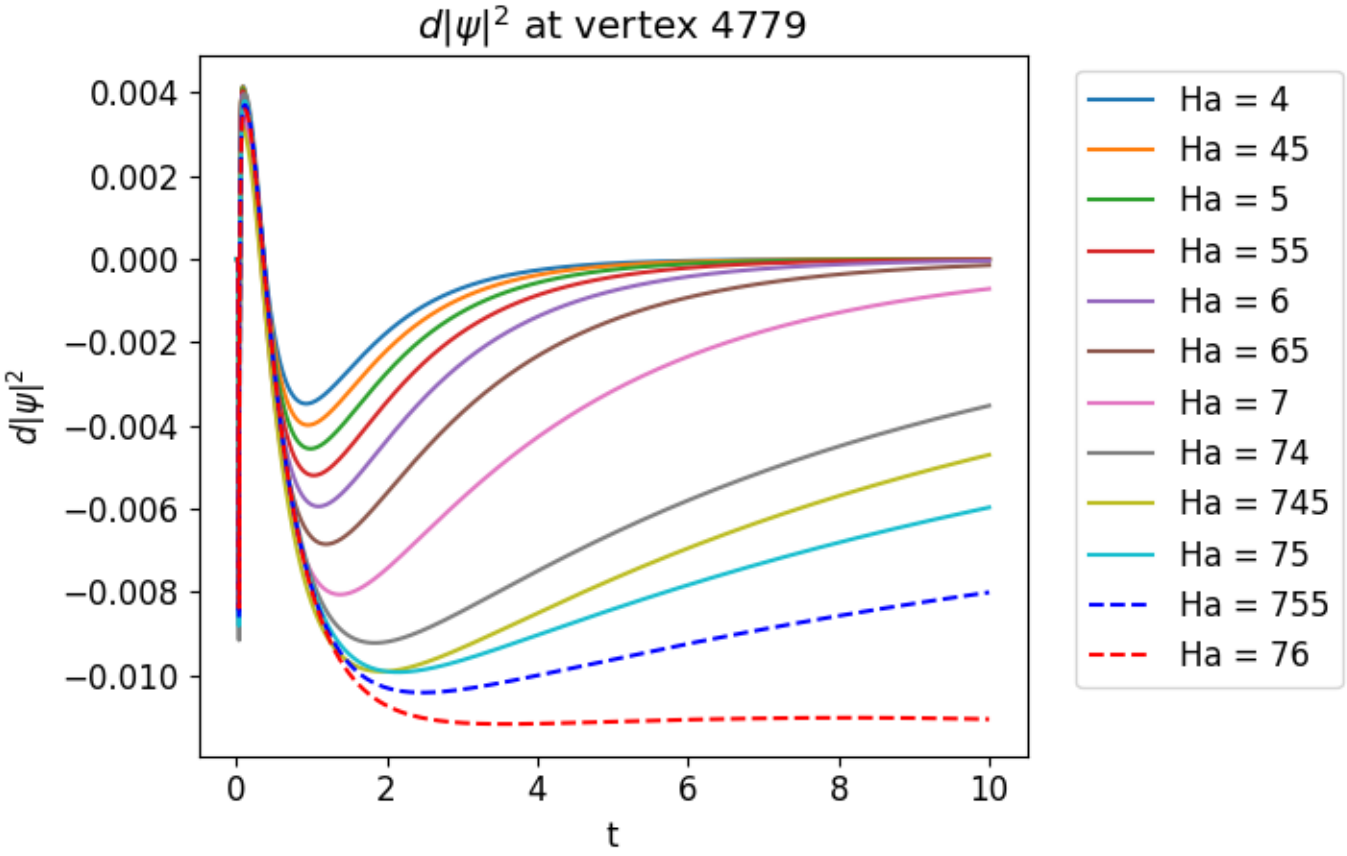


Figure 3.5 We observe the behavior of noise added to the order parameter at a given vertex and varying magnetic fields Ha. After a few time steps we introduce noise to the cylinder. Due to the interaction of this vertex with its neighbors we observe transient fluctuating behavior. For $t > 2$ the noise decays exponentially except for $Ha = 76$ which appears to stay constant.

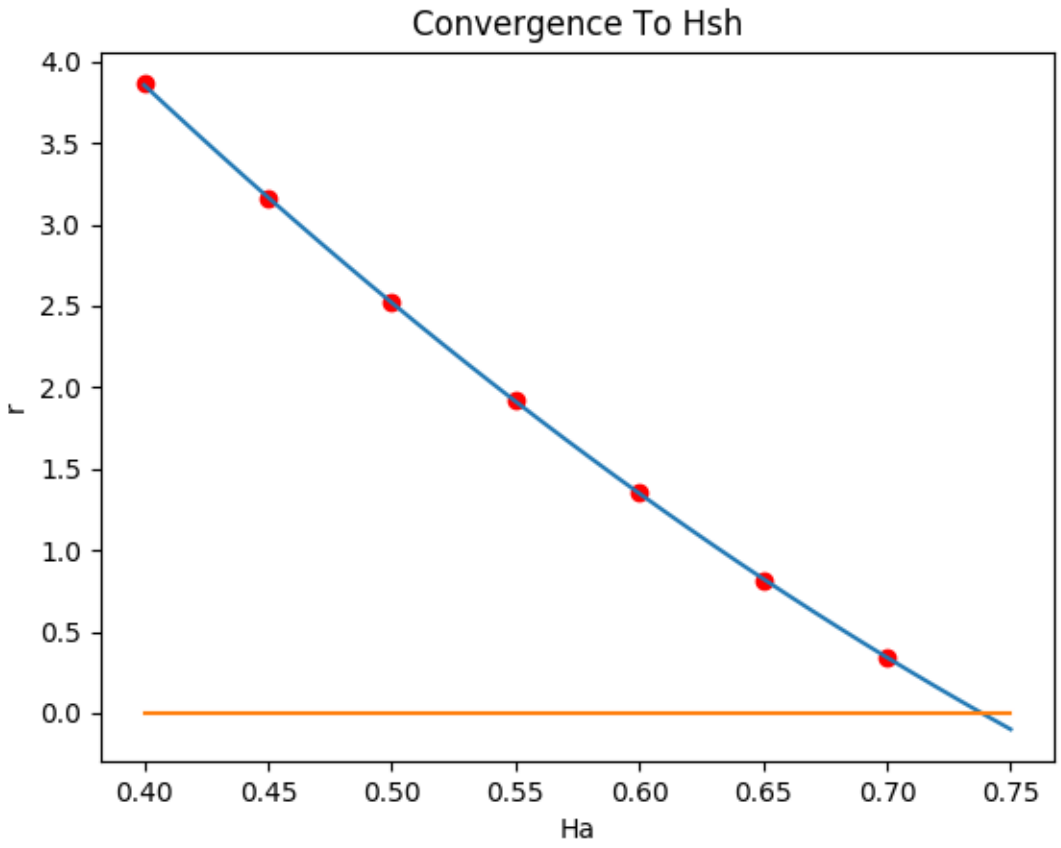


Figure 3.6 The red dots indicate various values of the bifurcation parameter at varying applied fields. A quadratic fit to this data yields a superheating field of 0.7383.

	p1	p2
dt = 0.1	0.73834	0.73564
dt = 0.05	0.73810	0.73565
dt = 0.01	0.73791	0.73528

Table 3.2 H_{sh} Variations

3.3 Surface Roughness

The previous results are useful in visualizing the critical wavelength and finding H_{sh}, but physical SRF cavities are not perfectly symmetric. Considering roughness on a cylinder takes us one step closer toward simulating SRF cavities. We now consider the rough cylinder mentioned in chapter 2. In Fig. 3.7 we see what happens when we set $Hmax = 0.7$ for a rough cylinder. The vortices no longer penetrate the surface uniformly. Instead they prefer to nucleate in the troughs of the surface. For this geometry $H_{sh} = 0.6866$ with $\kappa = 4$. This shows that adding roughness can lead to a decrease in H_{sh}.

3.4 Accuracy of Results

Li et al. have already performed an error analysis of several geometries for their formulation of the TDGL equations [18]. Our previous choices for the timestep and mesh density were based on balancing accuracy of expected physics and minimizing time to run simulations. We can increase mesh density and adjust the timestep until physical observables ($|\psi|^2$, H_{sh}) meet desired accuracy.

Table 3.2 shows how H_{sh} varies as we change the timestep and polynomial order of our finite elements. Keeping the mesh the same we can see that second order polynomials reduces H_{sh} and raising the step size increases H_{sh}. This table shows we are accurate to two significant figures.

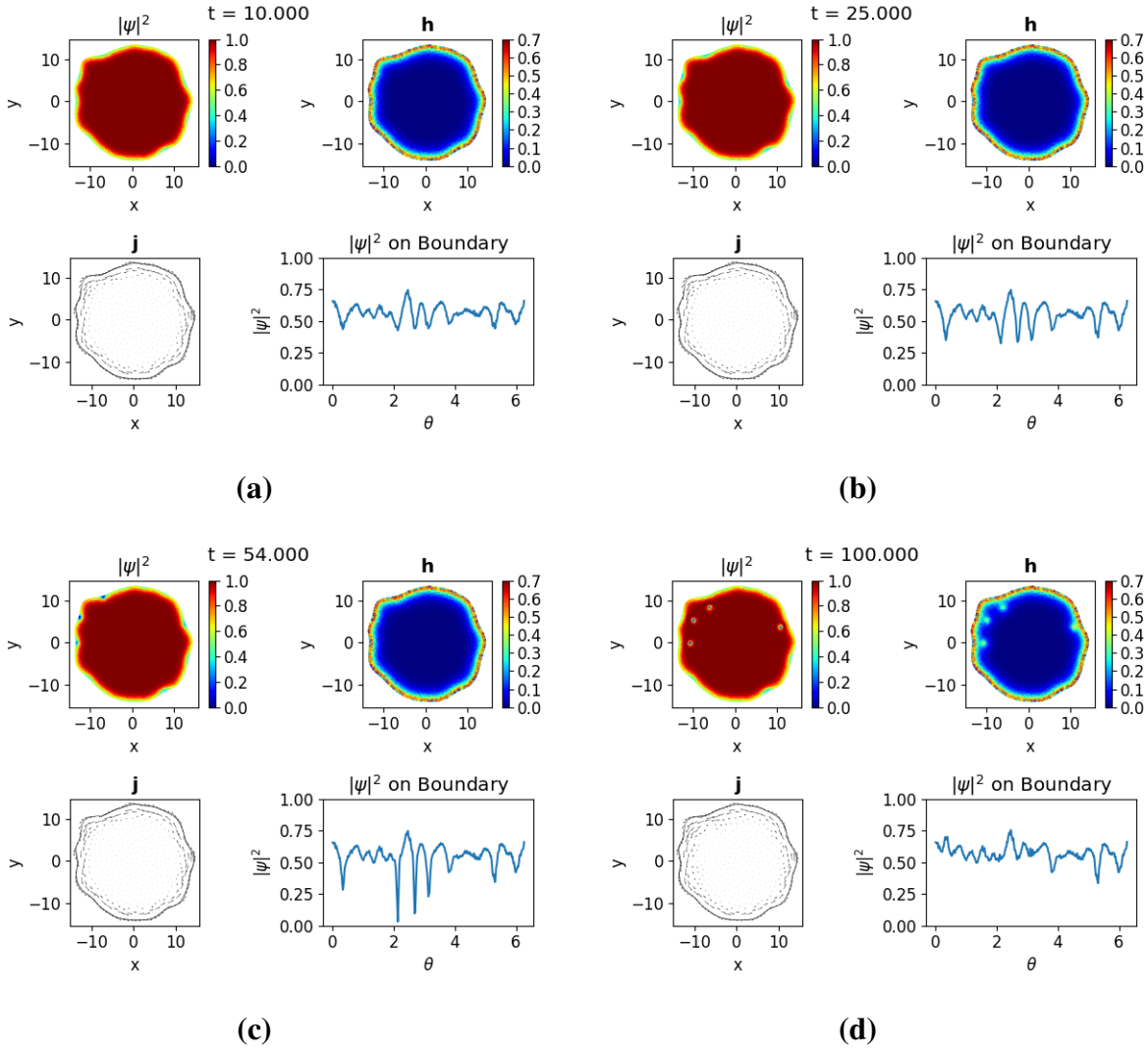


Figure 3.7 We raise the field to $Ha = 0.7$ on a rough cylinder. Unlike the symmetric cylinder the order parameter is not uniform before vortex nucleation. As the field increases the vortices form in the troughs of the surface.

We have also calculated the mean squared error between $|\psi|^2$ from the steady state solution of the time-dependent problem with $|\psi|^2$ from the time-independent problem with $Ha = 0.7$. The error is 1.2644e-05.

Based on these two results and the ability to reproduce expected physics we are confident our simulations are meaningful.

Chapter 4

Conclusion and Possible Extensions

We have created software that solves the TDGL equations to capture interesting physics in superconductors. We have seen how surface roughness and thermal fluctuations influence vortex dynamics on a cylindrical superconductor. Finally we demonstrated how to use saddle-node bifurcation theory to estimate H_{sh}.

Much of our efforts were dedicated to implementing FEM in FEniCS, debugging code, and evaluating the accuracy of our simulations. As interesting physics surfaced and collaborators suggested new ideas we found there are ways to continue this research. The following are potential extensions and applications of our methods.

4.1 Possible Extensions

Spatial Variations

We assumed no spatial variations in T_c or H_c. This is true of a material that contains the same quantity of atoms uniformly. In reality all superconducting materials are dirty. In a real SRF cavity there may be regions of nonsuperconducting material. Often grain boundaries can introduce

discontinuities. These inhomogeneities could be accounted for by letting κ vary spatially. We would have to rederive the GL equations and implement the changes in our software.

Surface Roughness

As part of our work described here, we analyzed how surface roughness on a cylinder lowered H_{sh} and how vortex nucleation changed. The next step is to create a database of meshes of varying roughness, evaluate their corresponding H_{sh}, and statistically infer when roughness matters.

Varying Magnetic Field Dynamics

We raised the field exponentially from zero so that initially the field increased rapidly but slowed down near H_{sh}. This allowed us to approach H_{sh} in a reasonable amount of time without forcing a transition due to rapid changes in magnetic fields. AC fields are used in SRF cavities. It would be informative to see how vortices enter and exit a superconducting material in an oscillating field.

Generalizing to 3-D

Our cylindrical geometry is limited to z-independent variations. The magnetic field must also be perpendicular to our circular cross section. There are some approaches to solving TDGL in 3-D [21]. They require complex finite elements and solvers to produce accurate results.

4.2 Conclusions

This work enables further study of many interesting phenomena near the superheating transition. We hope that this work will serve as a foundation for many future projects that can answer many questions that are beyond the scope of the current project. We will continue to collaborate with members of CBB and colleagues at Brigham Young University to explore this fascinating field.

Our source code that solves the TDGL equations is found in appendix A and code that compares the TDGL solutions with GL solutions in appendix B.

Appendix A

TDGL Code

The file *TDGL_Arbitrary_Long_Final.py* initializes the objects used in FEniCS to solve TDGL. I have included occasional comments to explain what each section of code does but much of it will require familiarity with FEniCS to make any sense.

```
import dolfin as d
import numpy as np
poly = 1

class HA(d.Expression):
    def __init__(self,t, ha, degree=None):
        self.t = t
        self.ha = ha
        # self.theta = np.
    def eval(self, values, x):
        values[0]=(self.ha)*(1.0 - np.exp(-self.t))
```

```
class DHADT(d.Expression):
    def __init__(self, t, ha, degree=None):
        self.t = t
        self.ha = ha
    def eval(self, values, x):
        values[0]=(self.ha)*(np.exp(-self.t))

class InitialFF(d.Expression):
    def eval(self, values, x):
        #Real
        values[0]=0.8
        #imaginary
        values[1]=0.6
    def value_shape(self):
        return (2,)

class InitialQ(d.Expression):
    def eval(self, values, x):
        values[0]=0.0
        values[1]=0.0
    def value_shape(self):
        return (2,)

def boundary(x, on_boundary):
```

```
    return on_boundary
class TDGL_Arbitrary:
    def __init__(self, ha, kappa, mesh_in, dt, parallel = False, comm_in=None):
        ##Create the mesh.
        self.ha = ha
        self.mesh = d.Mesh(mesh_in)
        self.meshsize = len(self.mesh.coordinates())
        ##Define the function spaces and the test and trial functions.
        self.F = d.FunctionSpace(self.mesh, "Lagrange", poly)
        self.Felem=d.FiniteElement("Lagrange", self.mesh.ufl_cell(),poly)
        self.FF = d.FunctionSpace(self.mesh, self.Felem * self.Felem)
        self.ff = d.Function(self.FF)
        self.ff0 = d.Function(self.FF)
        self.ff_init = InitialFF(degree=2)
        self.ff0.interpolate(self.ff_init)
        #Q
        self.Q = d.VectorFunctionSpace(self.mesh, "Lagrange", poly)
        self.q = d.Function(self.Q)
        self.q0 = d.Function(self.Q)
        self.q_init = InitialQ(degree=2)
        self.q0.interpolate(self.q_init)
        self.q_2 = d.Function(self.Q)
        self.qassigner = d.FunctionAssigner(self.Q, self.Q)
        self.qsubassigners = [d.FunctionAssigner(self.Q.sub(i), self.F)\
for i in range(2)]
```

```
#p, q2, u, v
self.p = d.Function(self.F)
self.p_2 = d.Function(self.F)
self.ptrial = d.TrialFunction(self.F)
self.q2 = d.Function(self.F)
self.q2trial = d.TrialFunction(self.F)
self.q2_2 = d.Function(self.F)
self.u = d.Function(self.F)
self.utrial = d.TrialFunction(self.F)
self.u_1 = d.Function(self.F)
self.u_2 = d.Function(self.F)
self.v = d.Function(self.F)
self.vtrial = d.TrialFunction(self.F)
self.v_1 = d.Function(self.F)
self.v_2 = d.Function(self.F)
self.ptest = d.TestFunction(self.F)
self.q2test = d.TestFunction(self.F)
self.utest = d.TestFunction(self.F)
self.vtest = d.TestFunction(self.F)
self.wtest = d.TestFunction(self.F)
self.w_1 = d.Function(self.F)
self.w_2 = d.Function(self.F)
self.wtrial = d.TrialFunction(self.F)
self.w = d.Function(self.F)
##Set initial time
```

```
self.t = 0.0
self.ts = [self.t]
self.dts = [0.0]
#define Ha and dHdt
self.Ha = HA(self.t, self.ha, degree=1)
self.dHdt = DHADT(self.t, self.ha, degree=1)
##Define the time step
self.dt = dt
self.dff = d.TrialFunction(self.FF)#This is used in calculating the jacobian
self.ffassigner = d.FunctionAssigner(self.FF, self.FF)
self.fftofrassigner = d.FunctionAssigner(self.F, self.FF.sub(0))
self.fftofiassigner = d.FunctionAssigner(self.F, self.FF.sub(1))
self.fassigner = d.FunctionAssigner(self.F, self.F)
self.fr, self.fi = d.split(self.ff)
self.frtest, self.fitest = d.TestFunctions(self.FF)
self.fr_1 = d.Function(self.F)
self.fi_1 = d.Function(self.F)
self.fr_2 = d.Function(self.F)
self.fi_2 = d.Function(self.F)
self.fftofrassigner.assign(self.fr_1,self.ff0.sub(0))
self.fftofiassigner.assign(self.fi_1,self.ff0.sub(1))
self.kappa = kappa
self.eta = 1.0
#Solve for initial u_1 and v_1
self.ubv = d.Constant(0.0)
```

```
        self.ubc = d.DirichletBC(self.F,self.ubv, boundary)
        self.uLHS0 = d.inner(d.grad(self.utrial),d.grad(self.utest))*d.dx
        self.uRHS0 = d.inner(self.q0,d.curl(self.utest))*d.dx
        #set up problem for v
        self.vLHS0 = d.inner(d.grad(self.vtrial),d.grad(self.vtest))*d.dx
        self.vRHS0 = d.inner(self.q0,d.grad(self.vtest))*d.dx
        ##solve for inital v and u
        d.solve(self.uLHS0 == self.uRHS0, self.u, self.ubc)
        self.fassigner.assign(self.u_1, self.u)
        d.solve(self.vLHS0 == self.vRHS0, self.v)
        self.fassigner.assign(self.v_1, self.v)
        #set up problem for f (psi)
        self.F1 = (-self.fi_1*self.fitest + self.fi*self.fitest - self.fr_1*self.frtes\
t + self.fr*self.frtest)*d.dx + self.dt/(self.kappa**2)*(d.inner(d.grad(self.fi),d.gra\
d(self.fitest)) + d.inner(d.grad(self.fr),d.grad(self.frtest)))*d.dx + self.dt/self.ka\
ppa*(d.inner(self.q0,d.grad(self.fr))*self.fitest - d.inner(self.q0,d.grad(self.fi))*s\
elf.frtest)*d.dx + self.dt/self.kappa*(-d.inner(self.q0,d.grad(self.fites\t)) *self.fr\
 + d.inner(self.q0,d.grad(self.frtest))*self.fi)*d.dx + self.dt*d.inner(self.q0,self.q\
0)*(self.fi*self.fitest + self.fr*self.frtest)*d.dx + self.dt*(-self.fi*self.fitest + \
self.fi_1**2*self.fi*self.fitest + self.fi*self.fr_1**2*self.fitest - self.fr*self.frt\
est + self.fi_1**2*self.fr*self.frtest + self.fr_1**2*self.fr*self.frtest)*d.dx + self\
.dt*self.eta*self.kappa*(d.inner(self.q0, d.grad(self.fr))*self.fitest - d.inner(self.\
q0, d.grad(self.fi))*self.frtest + d.inner(self.q0,d.grad(self.fitest))*self.fr - d.in\
ner(self.q0, d.grad(self.frt\est))*self.fi)*d.dx
        self.F2 = (self.fr_1*self.fitest - self.fr*self.fitest - self.fi_1*self.frtest\
```

```
 + self.fi*self.frtest)*d.dx + self.dt/(self.kappa**2)*(-d.inner(d.grad(self.fr),d.gra\
d(self.fitest)) + d.inner(d.grad(self.fi),d.grad(self.frtest)))*d.dx + self.dt/self.ka\
ppa*(d.inner(self.q0,d.grad(self.fi))*self.fitest + d.inner(self.q0,d.grad(self.fr))*s\
elf.frtest)*d.dx + self.dt/self.kappa*(-d.inner(self.q0,d.grad(self.fitest))*self.fi -\
 d.inner(self.q0,d.grad(self.frtest))*self.fr)*d.dx + self.dt*d.inner(self.q0,self.q0)\
*(-self.fr*self.fitest + self.fi*self.frtest)*d.dx + dt*(self.fr*self.fitest - self.fi\
_1**2*self.fr*self.fitest - self.fr*self.fr_1**2*self.fitest - self.fi*self.frtest + s\
elf.fi_1**2*self.fi*self.frtest + self.fr_1**2*self.fi*self.frtest)*d.dx + self.dt*sel\
f.eta*self.kappa*(d.inner(self.q0, d.grad(self.fi))*self.fitest + d.inner(self.q0, d.g\
rad(self.fr))*self.frtest + d.inner(self.q0,d.grad(self.fitest))*self.fi + d.inner(sel\
f.q0, d.grad(self.frtest))*self.fr)*d.dx
        self.Fs = self.F1+self.F2
        self.a1 = d.derivative(self.Fs, self.ff, self.dff)
        self.M = self.fr*d.dx()
#         ## This is how high the error will get before the solver starts refining the\
 mesh. We have set it high so it wont refine for the moment.
        self.tol=1e4
#         ## We have None for the boundary conditions because it is taken care of in t\
he weak form
        self.fproblem = d.NonlinearVariationalProblem(self.Fs, self.ff, None, self.a1)
        self.fsolver = d.NonlinearVariationalSolver(self.fproblem)
        ##set up problem for P
        self.pbv = d.Constant(0.0)
        # self.pbc = d.DirichletBC(self.F,self.pbv, By)
        self.pbc = d.DirichletBC(self.F,self.pbv, boundary)
```

```
        self.pRHS =d.inner(self.q0*(self.fi_1*self.fi+self.fr_1*self.fr)+1.0/self.kapp\
a*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*self.f\
r_1),d.curl(self.ptest))*d.dx
        self.pLHS = d.inner(d.grad(self.ptrial),d.grad(self.ptest))*d.dx
        self.err = []
        self.i = 0
        ##set up problem for q
        self.q2RHS =d.inner(self.q0*(self.fi_1*self.fi+self.fr_1*self.fr)+1.0/self.kap\
pa*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*self.fr_1),d.grad(self.q2test))*d.dx
        self.q2LHS = d.inner(d.grad(self.q2trial),d.grad(self.q2test))*d.dx
        ##set up problem for u
        self.uLHS = self.utrial*self.utest*d.dx + self.dt*d.inner(d.grad(self.utrial),\
d.grad(self.utest))*d.dx
        self.uRHS = self.u_1*self.utest*d.dx + self.dt*(self.Ha - self.p_2)*self.utest\
*d.dx
        #set up problem for v
        self.vLHS = self.vtrial*self.vtest*d.dx + self.dt*d.inner(d.grad(self.vtrial),\
d.grad(self.vtest))*d.dx
        self.vRHS = self.v_1*self.vtest*d.dx - self.dt*(self.q2_2)*self.vtest*d.dx
        #set up problem for w
        self.wLHS = self.wtrial*self.wtest*d.dx + self.dt*d.inner(d.grad(self.wtrial),\
d.grad(self.wtest))*d.dx
        self.wRHS = self.w_1*self.wtest*d.dx - self.dt*d.inner(self.q0*(self.fi_1*self\
.fi+self.fr_1*self.fr)+1.0/self.kappa*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*sel\
f.fr_1),d.curl(self.wtest))*d.dx#add dHadt term when considering time case
```

```
        #save sols
        self.frsols = [self.fr_1.compute_vertex_values()]
        self.fisols = [self.fi_1.compute_vertex_values()]
        self.qsols = [self.q0.compute_vertex_values()]
        self.usols = [self.u_1.compute_vertex_values()]
        self.vsols = [self.v_1.compute_vertex_values()]
        self.psols = [self.p.compute_vertex_values()]
        self.q2sols = [self.q2.compute_vertex_values()]
        self.bsols0 = d.project(self.Ha + self.w_1, self.F)
        self.bsols = [self.bsols0.compute_vertex_values()]
        self.hsols0 = d.project(self.q0.dx(0), self.Q).compute_vertex_values()[self.me\
shsize:]-d.project(self.q0.dx(1), self.Q).compute_ver\
tex_values()[:self.meshsize]
        self.hsols = [self.hsols0]
        self.haongrid = d.Function(self.F)
        self.haongrid.interpolate(self.Ha)
        self.hsurf = [max(self.haongrid.compute_vertex_values())]
        self.ts = [self.t]
        self.N = len(self.fi_1.compute_vertex_values())

    def TimeStep(self):
        self.i +=1
        self.t += self.dt
        self.Ha.t = self.t
        self.dHdt.t = self.t
```

```
        # self.fsolver.solve(self.tol)
        self.fsolver.solve()
        self.fftofrassigner.assign(self.fr_2,self.ff.sub(0))
        self.fftofiassigner.assign(self.fi_2,self.ff.sub(1))
        d.solve(self.pLHS == self.pRHS, self.p, self.pbc)
        d.solve(self.pLHS == self.pRHS, self.p, self.pbc)
        self.fassigner.assign(self.p_2, self.p)
        d.solve(self.q2LHS == self.q2RHS,self.q2)
        self.fassigner.assign(self.q2_2, self.q2)
        d.solve(self.uLHS == self.uRHS, self.u, self.ubc)
        self.fassigner.assign(self.u_2, self.u)
        d.solve(self.vLHS == self.vRHS, self.v)
        self.fassigner.assign(self.v_2, self.v)
        d.solve(self.wLHS == self.wRHS,self.w)
        self.fassigner.assign(self.w_2, self.w)
        #calculate q_2
        self.q_2x = d.project(self.u_2.dx(1) + self.v_2.dx(0),self.F)
        self.q_2y = d.project(-self.u_2.dx(0) + self.v_2.dx(1),self.F)
        comps = [self.q_2x, self.q_2y]
        [self.qsubassigners[i].assign(self.q_2.sub(i),comp) for i, comp in\
enumerate(comps)]
        #Append new save values
        self.frsols.append(self.fr_2.compute_vertex_values())
        self.fisols.append(self.fi_2.compute_vertex_values())
        self.qsols.append(self.q_2.compute_vertex_values())
```

```
        self.q2sols.append(self.q2_2.compute_vertex_values())
        self.psols.append(self.p_2.compute_vertex_values())
        self.vsols.append(self.v_2.compute_vertex_values())
        self.usols.append(self.u_2.compute_vertex_values())
        # self.fsqrsols.append(self.fr_2.compute_vertex_values()**2 +self.fi_2.compute\
_vertex_values()**2)
        self.bsols.append(self.w_2.compute_vertex_values()+d.project(self.Ha, self.F).\
compute_vertex_values())
        self.hsols.append(d.project(self.q_2.dx(0), self.Q).compute_vertex_values()[se\
lf.meshsize:]-d.project(self.q_2.dx(1), self.Q).compu\
te_vertex_values()[:self.meshsize])
        self.dts.append(self.dt)
        self.ts.append(self.t)
        #assign new values for next step
        self.fassigner.assign(self.fr_1,self.fr_2)
        self.fassigner.assign(self.fi_1,self.fi_2)
        self.qassigner.assign(self.q0, self.q_2)
        self.fassigner.assign(self.u_1, self.u_2)
        self.fassigner.assign(self.v_1, self.v_2)
        self.haongrid.interpolate(self.Ha)
        self.hsurf.append(max(self.haongrid.compute_vertex_values()))
```

A very similar file is adapted to start from a previous solution.

```
import dolfin as d
import numpy as np
```

```
from scipy.interpolate import bisplrep, bisplev
np.random.seed(10)

###
#Restart
###

class HA(d.Expression):
    def __init__(self, ha, degree=None):
        self.ha = ha
        # self.theta = np.
    def eval(self, values, x):
        values[0]=self.ha

def boundary(x, on_boundary):
    return on_boundary

def makeNoise(mesh):
    coords = mesh.coordinates()
    r = [np.sqrt(coords[i][0]**2+coords[i][1]**2) for i in range(len(coords))]
    thetas = [np.arctan2(coords[i][1], coords[i][0]) for i in range(len(coords))]
    numk = 100
    k0 = 1.0
    sigma = 100.0
    # np.random.seed(10)
```

```
    tvals = np.array(thetas)*0.0
    rvals = np.array(r)*0.0
    for i in range(numk):
        val = 1.0/np.sqrt(2.0*np.pi)*np.exp(-(i-k0)**2/sigma)
        tvals += (-1)**np.random.randint(0,2)*val*np.cos(i*np.array(thetas))
        tvals += (-1)**np.random.randint(0,2)*val*np.sin(i*np.array(thetas))
        rvals += (-1)**np.random.randint(0,2)*val*np.cos(i*np.array(r)*2*np.pi/15.0)
        rvals += (-1)**np.random.randint(0,2)*val*np.sin(i*np.array(r)*2*np.pi/15.0)
    tvals = tvals/float(numk)/2.0
    rvals = rvals/float(numk)/2.0
    tvals = tvals/np.linalg.norm(tvals)
    rvals = rvals/np.linalg.norm(rvals)
    return tvals*rvals



class TDGL_Arbitrary:
    def __init__(self, ha, kappa, mesh_in, dt, prev_in, nzs = 0.0):
        ##Create the mesh.
        self.nzs = nzs
        self.ha = ha
        self.mesh = d.Mesh(mesh_in)
        stuff=np.load(prev_in)
        pfr = stuff['fr'][-1]
        pfi = stuff['fi'][-1]
```

```
pu = stuff['u'][-1]
pv = stuff['v'][-1]
pq = stuff['q'][-1]
self.meshsize = len(self.mesh.coordinates())
##Define the function spaces and the test and trial functions.
self.F = d.FunctionSpace(self.mesh, "Lagrange", 1)
self.Felem=d.FiniteElement("Lagrange", self.mesh.ufl_cell(),1)
self.FF = d.FunctionSpace(self.mesh, self.Felem * self.Felem)
self.ff = d.Function(self.FF)
self.ff0 = d.Function(self.FF)
self.frtoffassigner = d.FunctionAssigner(self.FF.sub(0), self.F)
self.fitoffassigner = d.FunctionAssigner(self.FF.sub(1), self.F)
v2d= d.vertex_to_dof_map(self.F)
#Assign previous fr vals
sorted1 = zip(v2d,pfr)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
sorted3 = [float(i[1]) for i in sorted2]
fr0 = d.Function(self.F)
fr0.vector()[:] = np.array(sorted3)
#Assign previous fi values
sorted1 = zip(v2d,pfi)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
sorted3 = [float(i[1]) for i in sorted2]
```

```
        fi0 = d.Function(self.F)
        fi0.vector()[:] = np.array(sorted3)
        self.frtoffassigner.assign(self.ff0.sub(0),fr0)
        self.fitoffassigner.assign(self.ff0.sub(1),fi0)
        #Q
        self.Q = d.VectorFunctionSpace(self.mesh, "Lagrange", 1)
        self.q = d.Function(self.Q)
        self.q0 = d.Function(self.Q)
        v2d = d.vertex_to_dof_map(self.Q)
        v2d = v2d.reshape((-1,self.mesh.geometry().dim()))
        N1 = len(pfr)
        qx = pq[:N1]
        qy = pq[N1:]
        q1 = np.array([[qx[i],qy[i]] for i in range(len(qx))])
        for i in range(len(q1)):
            self.q0.vector()[v2d[i]]=q1[i]
        self.q_2 = d.Function(self.Q)
        self.qassigner = d.FunctionAssigner(self.Q, self.Q)
        self.qsubassigners = [d.FunctionAssigner(self.Q.sub(i), self.F)\
for i in range(2)]
        #p, q2, u, v
        self.p = d.Function(self.F)
        self.p_2 = d.Function(self.F)
        self.ptrial = d.TrialFunction(self.F)
        self.q2 = d.Function(self.F)
```

```
 self.q2trial = d.TrialFunction(self.F)
 self.q2_2 = d.Function(self.F)
 self.u = d.Function(self.F)
 self.utrial = d.TrialFunction(self.F)
 self.u_1 = d.Function(self.F)
 self.u_2 = d.Function(self.F)
 self.v = d.Function(self.F)
 self.vtrial = d.TrialFunction(self.F)
 self.v_1 = d.Function(self.F)
self.v_2 = d.Function(self.F)
 #Assign previous u values
 v2d= d.vertex_to_dof_map(self.F)
 sorted1 = zip(v2d,pu)
 sorted2 = list(sorted1)
 sorted2.sort(key=lambda tup: tup[0])
 sorted3 = [float(i[1]) for i in sorted2]
 self.u_1.vector()[:] = np.array(sorted3)
 #Assign previous v values
 sorted1 = zip(v2d,pv)
 sorted2 = list(sorted1)
 sorted2.sort(key=lambda tup: tup[0])
 sorted3 = [float(i[1]) for i in sorted2]
 # v_1 = d.Function(self.F)
 self.v_1.vector()[:] = np.array(sorted3)
 # v.vector()[:] = np.array(sorted3)
```

```
self.ptest = d.TestFunction(self.F)
self.q2test = d.TestFunction(self.F)
self.utest = d.TestFunction(self.F)
self.vtest = d.TestFunction(self.F)
self.wtest = d.TestFunction(self.F)
self.w_1 = d.Function(self.F)
self.w_2 = d.Function(self.F)
self.wtrial = d.TrialFunction(self.F)
self.w = d.Function(self.F)
##Set initial time
self.t = 0.0
self.ts = [self.t]
self.dts = [0.0]
#define Ha and dHdt
self.Ha = HA(self.ha, degree=1)
##Define the time step
self.dt = dt
self.dff = d.TrialFunction(self.FF)#This is used in calculating the jacobian
self.ffassigner = d.FunctionAssigner(self.FF, self.FF)
self.fftofrassigner = d.FunctionAssigner(self.F, self.FF.sub(0))
self.fftofiassigner = d.FunctionAssigner(self.F, self.FF.sub(1))
self.fassigner = d.FunctionAssigner(self.F, self.F)
self.fr, self.fi = d.split(self.ff)
self.frtest, self.fitest = d.TestFunctions(self.FF)
self.fr_1 = d.Function(self.F)
```

```
        self.fi_1 = d.Function(self.F)
        self.fr_2 = d.Function(self.F)
        self.fi_2 = d.Function(self.F)
        self.fftofrassigner.assign(self.fr_1,self.ff0.sub(0))
        self.fftofiassigner.assign(self.fi_1,self.ff0.sub(1))
        self.kappa = kappa
        self.eta = 1.0
        #Solve for initial u_1 and v_1
        self.ubv = d.Constant(0.0)
        self.ubc = d.DirichletBC(self.F,self.ubv, boundary)
        self.F1 = (-self.fi_1*self.fitest + self.fi*self.fitest - self.fr_1*self.frtest\
 + self.fr*self.frtest)*d.dx + self.dt/(self.kappa**2)*(d.inner(d.grad(self.fi),d.grad(\
self.fitest)) + d.inner(d.grad(self.fr),d.grad(self.frtest)))*d.dx + self.dt/self.kappa\
*(d.inner(self.q0,d.grad(self.fr))*self.fitest - d.inner(self.q0,d.grad(self.fi))*self.\
frtest)*d.dx + self.dt/self.kappa*(-d.inner(self.q0,d.grad(self.fitest)) *self.fr + d.i\
nner(self.q0,d.grad(self.frtest))*self.fi)*d.dx + self.dt*d.inner(self.q0,self.q0)*(sel\
f.fi*self.fitest + self.fr*self.frtest)*d.dx + self.dt*(-self.fi*self.fitest + self.fi_\
1**2*self.fi*self.fitest + self.fi*self.fr_1**2*self.fitest - self.fr*self.frtest + sel\
f.fi_1**2*self.fr*self.frtest + self.fr_1**2*self.fr*self.frtest)*d.dx + self.dt*self.e\
ta*self.kappa*(d.inner(self.q0, d.grad(self.fr))*self.fitest - d.inner(self.q0, d.grad(\
self.fi))*self.frtest + d.inner(self.q0,d.grad(self.fitest))*self.fr - d.inner(self.q0,\
 d.grad(self.frtest))*self.fi)*d.dx
        self.F2 = (self.fr_1*self.fitest - self.fr*self.fitest - self.fi_1*self.frtest \
+ self.fi*self.frtest)*d.dx + self.dt/(self.kappa**2)*(-d.inner(d.grad(self.fr),d.grad(\
self.fitest)) + d.inner(d.grad(self.fi),d.grad(self.frtest)))*d.dx + self.dt/self.kappa\
```

```
*(d.inner(self.q0,d.grad(self.fi))*self.fitest + d.inner(self.q0,d.grad(self.fr))*self.\
frtest)*d.dx + self.dt/self.kappa*(-d.inner(self.q0,d.grad(self.fitest))*self.fi - d.in\
ner(self.q0,d.grad(self.frtest))*self.fr)*d.dx + self.dt*d.inner(self.q0,self.q0)*(-sel\
f.fr*self.fitest + self.fi*self.frtest)*d.dx + dt*(self.fr*self.fitest - self.fi_1**2*s\
elf.fr*self.fitest - self.fr*self.fr_1**2*self.fitest - self.fi*self.frtest + self.fi_\
1**2*self.fi*self.frtest + self.fr_1**2*self.fi*self.frtest)*d.dx + self.dt*self.eta*s\
elf.kappa*(d.inner(self.q0, d.grad(self.fi))*self.fitest + d.inner(self.q0, d.grad(sel\
f.fr))*self.frtest + d.inner(self.q0,d.grad(self.fitest))*self.fi + d.inner(self.q0, d\
.grad(self.frtest))*self.fr)*d.dx
        self.Fs = self.F1+self.F2
        self.a1 = d.derivative(self.Fs, self.ff, self.dff)
        self.M = self.fr*d.dx()
        self.tol=1e4
#         ## We have None for the boundary conditions because it is taken care of in t\
he weak form
        self.fproblem = d.NonlinearVariationalProblem(self.Fs, self.ff, None, self.a1)
        self.fsolver = d.AdaptiveNonlinearVariationalSolver(self.fproblem, self.M)
        ##set up problem for P
        self.pbv = d.Constant(0.0)
        self.pbc = d.DirichletBC(self.F,self.pbv, boundary)
        self.pRHS =d.inner(self.q0*(self.fi_1*self.fi+self.fr_1*self.fr)+1.0/self.kapp\
a*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*self.f\
r_1),d.curl(self.ptest))*d.dx
        self.pLHS = d.inner(d.grad(self.ptrial),d.grad(self.ptest))*d.dx
        self.err = []
```

```
        self.i = 0
        ##set up problem for q
        self.q2RHS =d.inner(self.q0*(self.fi_1*self.fi+self.fr_1*self.fr)+1.0/self.kap\
pa*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*self.fr_1),d.grad(self.q2test))*d.dx
        self.q2LHS = d.inner(d.grad(self.q2trial),d.grad(self.q2test))*d.dx
        ##set up problem for u, note this differs from previous u and v setup
        self.uLHS = self.utrial*self.utest*d.dx + self.dt*d.inner(d.grad(self.utrial),\
d.grad(self.utest))*d.dx
        self.uRHS = self.u_1*self.utest*d.dx + self.dt*(self.Ha - self.p_2)*self.utest\
*d.dx
        #set up problem for v
        self.vLHS = self.vtrial*self.vtest*d.dx + self.dt*d.inner(d.grad(self.vtrial)\
,d.grad(self.vtest))*d.dx
        self.vRHS = self.v_1*self.vtest*d.dx - self.dt*(self.q2_2)*self.vtest*d.dx
        #set up problem for w
        self.wLHS = self.wtrial*self.wtest*d.dx + self.dt*d.inner(d.grad(self.wtrial)\
,d.grad(self.wtest))*d.dx
        self.wRHS = self.w_1*self.wtest*d.dx - self.dt*d.inner(self.q0*(self.fi_1*sel\
f.fi+self.fr_1*self.fr)+1.0/self.kappa*(d.grad(self.fr)*self.fi_1 - d.grad(self.fi)*s\
elf.fr_1),d.curl(self.wtest))*d.dx#add dHadt term when considering time case
        #save sols
        self.frsols = [self.fr_1.compute_vertex_values()]
        self.fisols = [self.fi_1.compute_vertex_values()]
        self.qsols = [self.q0.compute_vertex_values()]
        self.usols = [self.u_1.compute_vertex_values()]
```

```
        self.vsols = [self.v_1.compute_vertex_values()]
        self.psols = [self.p.compute_vertex_values()]
        self.q2sols = [self.q2.compute_vertex_values()]
        # self.fsqrsols = [self.fr_1.compute_vertex_values()**2 +self.fi_1.compute_\
vertex_values()**2]
        self.bsols0 = d.project(self.Ha + self.w_1, self.F)
        self.bsols = [self.bsols0.compute_vertex_values()]
        self.hsols0 = d.project(self.q0.dx(0), self.Q).compute_vertex_values()[sel\
f.meshsize:]-d.project(self.q0.dx(1), self.Q).compute_vertex_values()[:self.meshsize]
        self.hsols = [self.hsols0]
        self.haongrid = d.Function(self.F)
        self.haongrid.interpolate(self.Ha)
        self.hsurf = [max(self.haongrid.compute_vertex_values())]
        self.ts = [self.t]
        self.N = len(self.fi_1.compute_vertex_values())

    def TimeStep(self):
        self.i +=1
        self.t += self.dt
        self.fsolver.solve(self.tol)
        self.fftofrassigner.assign(self.fr_2,self.ff.sub(0))
        self.fftofiassigner.assign(self.fi_2,self.ff.sub(1))
        d.solve(self.pLHS == self.pRHS, self.p, self.pbc)
        self.fassigner.assign(self.p_2, self.p)
        d.solve(self.q2LHS == self.q2RHS,self.q2)
```

```
        self.fassigner.assign(self.q2_2, self.q2)
        d.solve(self.uLHS == self.uRHS, self.u, self.ubc)
        self.fassigner.assign(self.u_2, self.u)
        d.solve(self.vLHS == self.vRHS, self.v)
        self.fassigner.assign(self.v_2, self.v)
        d.solve(self.wLHS == self.wRHS,self.w)
        self.fassigner.assign(self.w_2, self.w)
        #calculate q_2
        self.q_2x = d.project(self.u_2.dx(1) + self.v_2.dx(0),self.F)
        self.q_2y = d.project(-self.u_2.dx(0) + self.v_2.dx(1),self.F)
        comps = [self.q_2x, self.q_2y]
        [self.qsubassigners[i].assign(self.q_2.sub(i),comp) for i, comp in\
 enumerate(comps)]
        #Append new save values
        self.frsols.append(self.fr_2.compute_vertex_values())
        self.fisols.append(self.fi_2.compute_vertex_values())
        self.qsols.append(self.q_2.compute_vertex_values())
        self.q2sols.append(self.q2_2.compute_vertex_values())
        self.psols.append(self.p_2.compute_vertex_values())
        self.vsols.append(self.v_2.compute_vertex_values())
        self.usols.append(self.u_2.compute_vertex_values())
        # self.fsqrsols.append(self.fr_2.compute_vertex_values()**2 +self.\
fi_2.compute_vertex_values()**2)
        self.bsols.append(self.w_2.compute_vertex_values()+d.project(self.\
Ha, self.F).compute_vertex_values())
```

```
        self.hsols.append(d.project(self.q_2.dx(0), self.Q).compute_vertex\
_values()[self.meshsize:]-d.project(self.q_2.dx(1), self.Q).compute_vertex\
_values()[:self.meshsize])
        self.dts.append(self.dt)
        self.ts.append(self.t)
        #assign new values for next step
        self.fassigner.assign(self.fr_1,self.fr_2)
        self.fassigner.assign(self.fi_1,self.fi_2)
        self.qassigner.assign(self.q0, self.q_2)
        self.fassigner.assign(self.u_1, self.u_2)
        self.haongrid.interpolate(self.Ha)
        self.hsurf.append(max(self.haongrid.compute_vertex_values()))
```

These files do not need to be accessed by the user to run simulations. I have created an interface that calls the previous script, runs simulations, and saves the result. This file *oneRunFinal.py* is what follows.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as tri
from dolfin import *
import imp

plt.set_cmap('jet')
```

```
restart = True

set_log_active(False) #gets rid of extra output

timeAdapt = True #
dt = 0.1
dts = '.1'
ha = 0.77
Has = '77'
steps = 4
kappa=4.0
rs = '15'
cs = '25'
pHas = Has
pdts = '.1'
mesh_in = "SymCylmeshr%sc%s.xml" % (rs, cs)
prev_in = 'ha%sr%sc%sdt%s.npz' % (pHas, rs, cs, pdts)

###
#Starting from zero
###
if restart == False:
    import TDGL_Arbitrary_Long_Final
    imp.reload(TDGL_Arbitrary_Long_Final)
```

```
    Arbitrary = TDGL_Arbitrary_Long_Final.TDGL_Arbitrary
    tdgl = Arbitrary(ha, kappa, mesh_in, dt)

###
#starting from previous solution
###
if restart == True:
    import RestartTDGL_Arbitrary_Long_Final
    imp.reload(RestartTDGL_Arbitrary_Long_Final)
    Arbitrary = RestartTDGL_Arbitrary_Long_Final.TDGL_Arbitrary
    tdgl = Arbitrary(ha, kappa, mesh_in, dt, prev_in)



###
#Solve problem
###

for i in range(steps):
    tdgl.TimeStep()
    print(i)
print("It's all good")

def savedata():
    np.savez(outFileName, fr = tdgl.frsols, fi = tdgl.fisols,  q = tdgl.qsols,\
 t = tdgl.ts, b = tdgl.bsols, dt = tdgl.dts, h = tdgl.hsols,hsurf = tdgl.hsurf,\
```

```
 u = tdgl.usols, v = tdgl.vsols)


###
#Save solution
###


outFileName = 'ha%sr%sc%sdt%s.npz' %(Has, rs, cs, dts)
savedata()
```

Appendix B

GL Code

The following script uses a previous steady state solution as an initial guess to the time-independent problem.

```
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.tri as tri

###used to plot
def mesh2triang(mesh):
    xy = mesh.coordinates()
    return tri.Triangulation(xy[:, 0], xy[:, 1], mesh.cells())

###plots scalar functions
def plot(obj):
```

```
    plt.clf()
    plt.gca().set_aspect('equal')
    plt.tripcolor(mesh2triang(mesh), obj, shading = 'gouraud')
    plt.clim(min(obj),max(obj))
    plt.colorbar()



class HA(Expression):
    def __init__(self, ha, degree=None):
        self.ha = ha
        # self.theta = np.
    def eval(self, values, x):
        values[0]=self.ha


def boundary(x, on_boundary):
    return on_boundary


pHas = '7'
ha = 0.7
pdts = '.1'
pdts1 = '.1'
rs = '15'
cs = '25'
mesh_in = "FindingHa/meshes/SymCylmeshr%sc%s.xml" % (rs, cs)
prev_in = 'r%sc%ssols/ha%sr%sc%sdt%s.npz' % (rs, cs, pHas, rs, cs, pdts)
```

```
prev_in1 = 'r%sc%ssols/ha%sr%sc%sdt%s.npz' % (rs, cs, pHas, rs, cs, pdts1)
kappa = 4.0


mesh = Mesh(mesh_in)
stuff=np.load(prev_in)
pfr = stuff['fr'][-1]
pfi = stuff['fi'][-1]
pu = stuff['u'][-1]
pv = stuff['v'][-1]
stuff1=np.load(prev_in1)
pfr1 = stuff['fr'][-1]
pfi1 = stuff['fi'][-1]
pu1 = stuff['u'][-1]
pv1 = stuff['v'][-1]
F = FunctionSpace(mesh, "Lagrange", 1)
Felem=FiniteElement("Lagrange", mesh.ufl_cell(),1)
Q = VectorFunctionSpace(mesh, "Lagrange",1)
Qelem = VectorElement("Lagrange",mesh.ufl_cell(),1)
MS = FunctionSpace(mesh, MixedElement([Felem, Felem, Qelem]))
ms = Function(MS)
mst = TestFunction(MS)
frtomsassigner = FunctionAssigner(MS.sub(0), F)
fitomsassigner = FunctionAssigner(MS.sub(1), F)
Atomsassigner = FunctionAssigner(MS.sub(2), Q)
v2d= vertex_to_dof_map(F)
```

```
        #Assign previous fr vals
sorted1 = zip(v2d,pfr)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
sorted3 = [float(i[1]) for i in sorted2]
fr0 = Function(F)
fr0.vector()[:] = np.array(sorted3)
        #Assign previous fi values
sorted1 = zip(v2d,pfi)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
sorted3 = [float(i[1]) for i in sorted2]
fi0 = Function(F)
fi0.vector()[:] = np.array(sorted3)

#Assign previous A values
v2d = vertex_to_dof_map(Q)
v2d = v2d.reshape((-1,mesh.geometry().dim()))
N1 = len(pfr)
        #Assign previous u values
u0 = Function(F)
v2d= vertex_to_dof_map(F)
sorted1 = zip(v2d,pu)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
```

```
sorted3 = [float(i[1]) for i in sorted2]
u0.vector()[:] = np.array(sorted3)
        #Assign previous v values
v0 = Function(F)
sorted1 = zip(v2d,pv)
sorted2 = list(sorted1)
sorted2.sort(key=lambda tup: tup[0])
sorted3 = [float(i[1]) for i in sorted2]
v0.vector()[:] = np.array(sorted3)

Ax0 = project(u0.dx(1) + v0.dx(0),F)
Ay0 = project(-u0.dx(0) + v0.dx(1),F)
comps = [Ax0, Ay0]
A0 = Function(Q)
qsubassigners = [FunctionAssigner(Q.sub(i), F) for i in range(2)]
[qsubassigners[i].assign(A0.sub(i),comp) for i, comp in enumerate(comps)]

Ha = HA(ha, degree=1)
frtomsassigner.assign(ms.sub(0),fr0)
fitomsassigner.assign(ms.sub(1),fi0)
Atomsassigner.assign(ms.sub(2),A0)



(fr,fi,A) = split(ms)
(vr,vi,dA) = split(mst)
```

```
F1 = (2/kappa**2*inner(grad(fr),grad(vr)) - 2/kappa*inner(grad(fi),A)*vr + 2/kappa\
*fi*inner(A,grad(vr)) - 2*fr*vr + 2*inner(A,A)*fr*vr + 2*fi**2*fr*vr + 2*vr*fr**3)\
/2*dx

F2 = (2/kappa**2*inner(grad(fi),grad(vi)) + 2/kappa*inner(grad(fr),A)*vi - 2/kappa\
*fr*inner(A,grad(vi)) - 2*fi*vi + 2*inner(A,A)*fi*vi + 2*\
fr**2*fi*vi + 2*vi*fi**3)/2*dx

y = Expression('x[1]',degree=1)
x = Expression('x[0]', degree=1)
r = Expression('pow(x[0],2)+pow(x[1],2)',degree=1)

F3 = ((-A[0].dx(1)+A[1].dx(0))*(-dA[0].dx(1)+dA[1].dx(0)) - Ha*(-dA[0].dx(1)+dA[1]\
.dx(0)) + 1/kappa*fi*inner(grad(fr),dA) - 1/kappa*fr*inner(grad(fi),dA) + fi**2*in\
ner(A,dA) + fr**2*inner(A,dA))*dx + ((-A[0].dx(1)+A[1].dx(0))-Ha)*(dA[0]*y/r-dA[1]\
*x/r)*ds



Fs = F1+F2+F3

dms = TrialFunction(MS)
a1 = derivative(Fs, ms,dms)
```

```
problem = NonlinearVariationalProblem(Fs, ms, None, a1)
solver = NonlinearVariationalSolver(problem)
solver.solve()

frsol = ms.compute_vertex_values()[:N1]
fisol = ms.compute_vertex_values()[N1:2*N1]
Asol = ms.compute_vertex_values()[2*N1:]
H = project(-A[0].dx(1)+A[1].dx(0),F)



###used to plot
def mesh2triang(mesh):
    xy = mesh.coordinates()
    return tri.Triangulation(xy[:, 0], xy[:, 1], mesh.cells())

###plots scalar functions
def plot(obj):
    plt.clf()
    plt.gca().set_aspect('equal')
    plt.tripcolor(mesh2triang(mesh), obj, shading = 'gouraud')
    plt.clim(min(obj),max(obj))
    plt.colorbar()
```

Bibliography

[1] M. Sunde, L. C. Serpell, M. Bartlam, P. E. Fraser, M. B. Pepys, and C. C. Blake, "Common core structure of amyloid fibrils by synchrotron X-ray diffraction," Journal of molecular biology **273,** 729–739 (1997).

[2] C. R. Wie, "High resolution X-ray diffraction characterization of semiconductor structures," Materials Science and Engineering: R: Reports **13,** 1–56 (1994).

[3] J. Kortright, D. Awschalom, J. Stöhr, S. Bader, Y. Idzerda, S. Parkin, I. K. Schuller, and H.-C. Siegmann, "Research frontiers in magnetic materials at soft X-ray synchrotron radiation facilities," Journal of Magnetism and Magnetic Materials **207,** 7–44 (1999).

[4] H. Padamsee, *RF superconductivity: science, technology and applications* (John Wiley & Sons, 2009).

[5] S. Posen, N. Valles, and M. Liepe, "Radio Frequency Magnetic Field Limits of Nb and Nb 3 Sn," Physical review letters **115,** 047001 (2015).

[6] S. Posen and M. Liepe, "Advances in development of Nb 3 Sn superconducting radio-frequency cavities," Physical Review Special Topics-Accelerators and Beams **17,** 112001 (2014).

[7] R. Parks and M. Tinkham, "Superconductivity Vols. 1, 2," Physics Today **23,** 66 (1970).

[8] H. K. Onnes, "The superconductivity of mercury," Comm. Phys. Lab. Univ. Leiden **122,** 124 (1911).

[9] Q. Du, M. D. Gunzburger, and J. S. Peterson, "Analysis and approximation of the Ginzburg–Landau model of superconductivity," Siam Review **34,** 54–81 (1992).

[10] L. Gor'Kov and G. Eliashberg, "Generalization of the Ginzburg-Equations for Non-Stationary Problems in the Case of Alloys with Paramagnetic Impurities," 30 Years of the Landau Institute-Selected Papers. Edited by KHALATNIKOV ISAAK M ET AL. Published by World Scientific Publishing Co. Pte. Ltd., 1996. ISBN# 9789814317344, pp. 16-22 pp. 16–22 (1996).

[11] C. Bean and J. Livingston, "Surface barrier in type-II superconductors," Physical Review Letters **12,** 14 (1964).

[12] M. "Transtrum, "personal communication".

[13] M. K. Transtrum, G. Catelani, and J. P. Sethna, "Superheating field of superconductors within Ginzburg-Landau theory," Physical Review B **83,** 094505 (2011).

[14] S. H. Strogatz, *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering* (Westview press, 2014).

[15] T. J. Hughes, *The finite element method: linear static and dynamic finite element analysis* (Courier Corporation, 2012).

[16] S. J. Chapman, S. D. Howison, and J. R. Ockendon, "Macroscopic models for superconductivity," Siam Review **34,** 529–560 (1992).

[17] Q. Du, "Finite element methods for the time-dependent Ginzburg-Landau model of superconductivity," Computers & Mathematics with Applications **27,** 119–133 (1994).

[18] B. Li and Z. Zhang, "A new approach for numerical simulation of the time-dependent Ginzburg–Landau equations," Journal of Computational Physics **303,** 238–250 (2015).

[19] B. Li and Z. Zhang, "Mathematical and numerical analysis of the time-dependent Ginzburg–Landau equations in nonconvex polygons based on Hodge decomposition," Mathematics of Computation **86,** 1579–1608 (2017).

[20] A. Logg, K.-A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method: The FEniCS book* (Springer Science & Business Media, 2012), Vol. 84.

[21] H. Gao and W. Sun, "An efficient fully linearized semi-implicit Galerkin-mixed FEM for the dynamical Ginzburg–Landau equations of superconductivity," Journal of Computational Physics **294,** 329–345 (2015).